Università del Piemonte Orientale

Dipartimento di Scienze e Innovazione Tecnologica

Corso di Laurea Magistrale in Informatica

Tesi di Laurea

Public Pressure: Progettazione e Sviluppo di un Marketplace di NFT Musicali Multi-Blockchain

Relatrice:

Prof.ssa Giuliana Annamaria Franceschinis

Correlatrice:

Prof.ssa Lavinia Egidi

Candidato:

Sergiu Ioan Burla

Anno Accademico 2024/2025

Abstract

La presente tesi descrive il lavoro svolto durante un tirocinio presso l'azienda Efebia s.r.l. di Milano, nell'ambito della progettazione e dello sviluppo di **Public Pressure**, una piattaforma per la compravendita di musica digitale sotto forma di NFT (Non-Fungible Token).

Public Pressure si propone come punto di incontro tra artisti e appassionati di musica: gli artisti possono pubblicare i propri brani come oggetti digitali unici o in edizione limitata, fissarne il prezzo o metterli all'asta, e continuare a ricevere una percentuale su ogni successiva rivendita. I collezionisti, dal canto loro, possono acquistare, detenere e rivendere questi brani con la garanzia di autenticità e provenienza che solo la blockchain può offrire.

Una delle principali sfide affrontate è stata rendere la piattaforma accessibile a un pubblico ampio: è possibile acquistare con carta di credito come su qualsiasi altro negozio online, oppure utilizzare criptovalute, su blockchain differenti. Garantire che due acquirenti non possano aggiudicarsi lo stesso esemplare unico pagando con metodi diversi ha richiesto la progettazione di un meccanismo di prenotazione temporanea e di sincronizzazione in tempo reale con i dati registrati sulla blockchain.

Il documento illustra le scelte progettuali e le soluzioni implementative adottate, dal funzionamento interno della piattaforma alla gestione dei pagamenti, dagli smart contract che regolano le aste fino all'infrastruttura cloud su cui il sistema è in esercizio. Viene inoltre descritto il processo di sviluppo, condotto con metodologia Agile e con un'attenzione particolare alla qualità del codice attraverso una suite di test automatizzati.

Parole chiave: NFT, blockchain, marketplace, musica digitale, smart contract, pagamenti digitali, multi-blockchain.

This thesis documents the work carried out during an internship at Efebia s.r.l. (Milan), focused on the design and development of **Public Pressure**, a platform for buying and selling music as NFTs (Non-Fungible Tokens).

Public Pressure connects artists and music enthusiasts: artists can release their tracks as unique or limited-edition digital items, set a fixed price or open an auction, and automatically receive a share of every future resale. Collectors can buy, hold, and resell these items with the authenticity and provenance guarantees that only a blockchain can provide.

One of the core challenges was making the platform accessible to a broad audience. Users can pay by credit card just as on any other online shop, or use cryptocurrency across multiple blockchains. Ensuring that two buyers cannot claim the same unique item through different payment methods required designing a temporary reservation mechanism and a real-time synchronisation service with the data recorded on-chain.

The thesis presents the design decisions and implementation choices made throughout the project — from the inner workings of the platform and payment handling, to the smart contracts governing auctions and the cloud infrastructure on which the system runs. The development process is also described, conducted with an Agile methodology and a strong focus on code quality through an extensive automated test suite.

Keywords: NFT, blockchain, marketplace, digital music, smart contracts, digital payments, multi-blockchain.

Indice

Abstract **1**

Elenco delle Figure **7**

Elenco delle Tabelle **8**

Glossario **9**

1 Introduzione **12**

1.1 Contesto e Rilevanza 12
1.2 Panoramica della Piattaforma 13
1.3 Dichiarazione del Problema 15
1.4 Scopo e Obiettivi della Tesi 16
1.5 Contesto del Tirocinio 17
1.6 Struttura della Tesi 18

2 Contesto Teorico **20**

2.1 La Tecnologia Blockchain 20
2.2 Non-Fungible Token (NFT) 34
2.3 NFT nel Settore Musicale 38
2.4 Meccanismo di Asta GBM 40

3 Tecnologie e Strumenti Utilizzati **41**

3.1 Linguaggi di Programmazione 42
3.2 Framework e Runtime 42
3.3 Database 44
3.4 Servizi Cloud AWS 45
3.5 IPFS e Archiviazione dei Metadati 46
3.6 Containerizzazione e Orchestrazione 48
3.7 Strumenti di Sviluppo e Collaborazione 49

4 Architettura del Sistema **52**
4.1 Architettura orientata ai servizi (SOA) 52
4.2 Modello dei Dati 56
4.3 Flusso Operativo del Sistema 59

5 Implementazione del Backend **62**
5.1 Struttura dell'Applicazione Fastify 62
5.2 Sistema di Autenticazione e Autorizzazione 65
5.3 Validazione dei Dati e JSON Schema 67
5.4 Gestione degli Ordini 69
5.5 Sistema di Code di Messaggi 70
5.6 Organizzazione degli Endpoint API 72
5.7 Content Management System 74

6 Smart Contract e Integrazione Blockchain **75**
6.1 Architettura dei Contratti 75
6.2 Implementazione del Diamond Pattern (EIP-2535) 76
6.3 Contratto GBM: Meccanismo di Asta Gamificato 79
6.4 Contratto TokenSale: Vendita a Prezzo Fisso 82
6.5 Contratto ERC721Multiple 82
6.6 Sistema di Royalty 84
6.7 Servizio di Blockchain-Pulling 86

7 Gestione dei Pagamenti **89**
7.1 Panoramica dei Gateway di Pagamento 89
7.2 Template Curated 91
7.3 Integrazione con Stripe 92
7.4 Integrazione con Coinbase Commerce 93
7.5 Pagamenti Diretti in ERC-20 (USDC) 94
7.6 Il Problema della Concorrenza 94
7.7 Disponibilità dei Gateway per Paese 97
7.8 Costi e Tempi delle Transazioni On-Chain 99

8 Testing e Qualità del Software **100**
8.1 Approccio Test-Driven Development 100
8.2 Infrastruttura di Test 101
8.3 Tipi di Test Implementati 103
8.4 Mock e Simulazione 105
8.5 Copertura del Codice 107

8.6 Integrazione nella Pipeline CI/CD 108
8.7 Test in Ambiente Reale 109

9 Deploy, CI/CD e Infrastruttura Cloud 110
9.1 Ambiente di Produzione 110
9.2 Pipeline CI/CD con GitHub Actions 112
9.3 Deploy con AWS Copilot 113
9.4 Infrastructure as Code con Terraform 116
9.5 Monitoraggio e Logging 117
9.6 Processo di Sviluppo Locale 118

10 Sicurezza e Protezione dei Dati 119
10.1 Sicurezza Applicativa 119
10.2 Sicurezza Infrastrutturale 121
10.3 Sicurezza degli Smart Contract 122
10.4 Protezione dei Dati Utente 123
10.5 Penetration Testing 124

11 Conclusioni e Sviluppi Futuri 125
11.1 Risultati Ottenuti 125
11.2 Sfide Incontrate 127
11.3 Competenze Acquisite 128
11.4 Sviluppi Futuri 129
11.5 Considerazioni Finali 130

Ringraziamenti 134

Elenco delle figure

1.1 Schema ad alto livello dell'interazione tra i componenti di Public Pressure. 14

2.1 Struttura di un albero di Merkle con quattro transazioni. La Merkle root viene inserita nell'header del blocco. 23
2.2 Le transazioni convergono nel mempool e vengono incluse nei blocchi in ordine seriale. Non esiste esecuzione parallela nell'EVM. 29
2.3 Architettura del Diamond Pattern (EIP-2535): il proxy inoltra le chiamate ai facet appropriati tramite delegatecall, condividendo lo storage. 38
2.4 Flusso di un'asta GBM: ogni offerente superato riceve un incentivo. Il meccanismo Hammer Time estende la durata se arrivano offerte negli ultimi istanti. 40

3.1 Stack tecnologico di Public Pressure: al centro il backend, con le sue dipendenze verso database, smart contract e provider di pagamento. . 41

4.1 Architettura a servizi di Public Pressure con i flussi di comunicazione via SQS e MongoDB condiviso. 54
4.2 Macchina a stati del modello Template. 57
4.3 Macchina a stati del modello Order. 58
4.4 Diagramma ER semplificato delle relazioni tra le entità principali del database. 59
4.5 Flusso sequenziale dell'acquisto di un NFT a prezzo fisso, dalla richiesta iniziale al completamento. 61

5.1 Ciclo di vita di una richiesta nel backend Fastify, attraverso i layer di sicurezza, autenticazione, validazione e risposta. 63
5.2 Composizione dei middleware di autenticazione. 67

6.1 Flusso di una chiamata attraverso il Diamond: il selettore della funzione determina il facet di destinazione, che viene eseguito nel contesto di storage del Diamond tramite delegatecall. 77

6.2 Pipeline di processamento del servizio di blockchain-pulling: i blocchi vengono recuperati dal nodo RPC, gli eventi vengono filtrati e instradati ai processori specializzati, che aggiornano il database in modo atomico. 88

7.1 Confronto dei flussi di pagamento per i tre gateway principali. I tempi di conferma variano significativamente tra FIAT e crypto. 90
7.2 Gestione della concorrenza: la prenotazione atomica al tempo t_0 garantisce che ogni edizione sia assegnata a un solo utente, indipendentemente dai tempi di conferma del pagamento. 95

8.1 Piramide dei test di Public Pressure: la base è costituita dai test unitari e di integrazione, i test più costosi sono in cima. 101
8.2 Pipeline CI/CD con 4 shard di test paralleli: il deploy è condizionato al successo di tutti gli shard e del linting. 108

9.1 Topologia dell'infrastruttura AWS di produzione: i servizi ECS Fargate operano in subnet private, esposti tramite ALB. 110

10.1 I tre livelli di sicurezza di Public Pressure: infrastruttura cloud, applicazione backend e smart contract on-chain. 119

Elenco delle tabelle

2.1 Costo in gas di operazioni EVM comuni (valori post-EIP-3529) . . . 30

3.1 Pacchetti principali della repository 50

4.1 Dimensioni del progetto Public Pressure 52
4.2 Code SQS generiche 55
4.3 Code SQS per blockchain (replicate per ogni rete) 55

5.1 Moduli API del backend, raggruppati per area funzionale 73

7.1 Confronto dei gateway di pagamento supportati 89
7.2 Costi indicativi delle operazioni on-chain per blockchain (in USD, a prezzi gas medi) 99

8.1 Copertura dei test di integrazione del backend 104
8.2 Riepilogo complessivo della suite di test 107
8.3 Metriche di copertura del codice (backend) 107

9.1 Workflow CI/CD per servizio e ambiente 112
9.2 Differenze di configurazione tra gli ambienti 117

11.1 Indicatori quantitativi del sistema in produzione 126

Glossario

ALB Application Load Balancer — servizio AWS di bilanciamento del carico a livello applicativo.

API Application Programming Interface — interfaccia per la comunicazione tra componenti software.

CDN Content Delivery Network — rete di distribuzione dei contenuti.

CEI Checks-Effects-Interactions — pattern di sicurezza per smart contract che previene attacchi di reentrancy.

CI/CD Continuous Integration / Continuous Deployment — pratica di automazione del build, test e deploy.

CRUD Create Read Update Delete — funzioni di base per la manipolazione di dati.

DApp Decentralized Application — applicazione che interagisce con smart contract su blockchain.

DLQ Dead-Letter Queue — coda che raccoglie i messaggi non elaborati con successo.

ECR Elastic Container Registry — servizio AWS di orchestrazione di container Docker.

iiì

ECS Elastic Container Service — servizio AWS di gestione di immagini Docker.

EIP Ethereum Improvement Proposal — proposta di miglioramento per il protocollo Ethereum.

EOA Externally Owned Account — account Ethereum controllato da una chiave privata.

ERC Ethereum Request for Comments — standard tecnico per l'ecosistema Ethereum.

ERC-20 Standard per token fungibili su Ethereum (es. USDC, USDT).

ERC-721 Standard per Non-Fungible Token su Ethereum.

ERC-1155 Standard multi-token che supporta sia token fungibili che non fungibili.

ERC-2981 Standard per le royalty sulle vendite secondarie di NFT.

EVM Ethereum Virtual Machine — macchina virtuale che esegue il bytecode degli smart contract.

FIAT Valuta tradizionale emessa da un governo (es. EUR, USD).

GBM Gamified Bidding Mechanism — meccanismo di asta in cui gli offerenti superati ricevono un incentivo.

IAM Identity and Access Management — servizio AWS per la gestione di permessi e ruoli.

IaC Infrastructure as Code — gestione dell'infrastruttura tramite file di configurazione.

IPFS InterPlanetary File System — protocollo di storage decentralizzato peer-to-peer.

JWT JSON Web Token — standard per l'autenticazione tramite token firmati.

KMS Key Management Service — servizio AWS per la gestione delle chiavi crittografiche.

KYC Know Your Customer — processo di verifica dell'identità degli utenti.

NFT Non-Fungible Token — token digitale unico e non divisibile registrato su blockchain.

ODM Object Document Mapper — libreria che mappa oggetti del linguaggio a documenti del database.

OTP One-Time Password — codice monouso per l'autenticazione a due fattori.

RPC Remote Procedure Call — protocollo per l'invocazione remota di funzioni su un nodo blockchain.

SOA Service-Oriented Architecture — architettura in cui le funzionalità sono distribuite su servizi indipendenti.

SQS Simple Queue Service — servizio AWS di code di messaggi.

TDD Test-Driven Development — metodologia in cui i test vengono scritti prima del codice.

TLS Transport Layer Security — protocollo crittografico per la sicurezza delle comunicazioni.

USDC USD Coin — stablecoin ancorata al dollaro statunitense, conforme a ERC-20.

UUPS Universal Upgradeable Proxy Standard — pattern di aggiornabilità degli smart contract.

VPC Virtual Private Cloud — rete virtuale isolata all'interno dell'infrastruttura AWS.

2FA Two-Factor Authentication — autenticazione a due fattori.

Capitolo 1

Introduzione

1.1 Contesto e Rilevanza

L'industria musicale ha attraversato profondi cambiamenti negli ultimi decenni, passando dalla distribuzione fisica dei supporti (vinili, CD, cassette) alla distribuzione digitale tramite piattaforme di streaming come Spotify, Apple Music e Tidal. Tuttavia, questo modello ha sollevato critiche significative riguardo alla remunerazione degli artisti: secondo un rapporto della CISAC (Confédération Internationale des Sociétés d'Auteurs et Compositeurs), gli artisti ricevono in media meno di un centesimo di dollaro per singolo ascolto sulle principali piattaforme di streaming [15].

In questo contesto, l'emergere della tecnologia blockchain e dei Non-Fungible Token (NFT) ha aperto nuove prospettive per la monetizzazione dei contenuti musicali. Gli NFT rappresentano token digitali unici e indivisibili, registrati su blockchain, che possono fungere da prova di autenticità e proprietà di un bene digitale. A differenza dei token fungibili come Bitcoin o Ether, ogni NFT è unico e può rappresentare un oggetto specifico: un brano musicale, un album, un'opera d'arte visiva collegata alla musica, o persino esperienze esclusive offerte dagli artisti ai loro fan.

Il mercato degli NFT ha conosciuto una crescita esponenziale tra il 2020 e il 2022, raggiungendo un volume di scambi globale superiore ai 25 miliardi di dollari nel 2021 secondo i dati di DappRadar [14]. Sebbene gran parte dell'attenzione mediatica si sia concentrata sugli NFT artistici e sui "collectible" digitali, il settore musicale ha iniziato a esplorare attivamente le potenzialità di questa tecnologia. Artisti come Kings of Leon, 3LAU e Grimes hanno pubblicato album e brani come NFT, generando milioni di dollari in vendite dirette e dimostrando la vitalità di questo nuovo canale di distribuzione.

Questo progetto di tesi si inserisce in tale contesto, concentrandosi sulla progettazione e lo sviluppo di **Public Pressure**, un marketplace specializzato nella compravendita di

NFT musicali. Public Pressure si distingue dai marketplace generalisti (come OpenSea o Rarible) per tre caratteristiche fondamentali:

1. **Specializzazione verticale nel settore musicale**: la piattaforma è progettata specificamente per artisti musicali, etichette discografiche e collezionisti di musica, con funzionalità dedicate come la gestione delle royalty, la categorizzazione per generi musicali e il supporto a diversi tipi di media (audio, video, collectible).

2. **Supporto multi-blockchain**: a differenza della maggior parte dei marketplace che operano su una singola blockchain (tipicamente Ethereum), Public Pressure supporta simultaneamente diverse blockchain Ethereum-based, tra cui Moonbeam (ecosistema Polkadot), Polygon, Base e Astar, offrendo agli utenti flessibilità nella scelta della rete in base a costi di transazione, velocità e preferenze personali.

3. **Pagamenti ibridi FIAT/crypto**: la piattaforma consente l'acquisto di NFT sia tramite carta di credito/debito (attraverso Stripe) sia tramite criptovalute (Coinbase Commerce e pagamenti diretti in USDC), rendendo il marketplace accessibile anche a utenti non esperti di criptovalute.

1.2 Panoramica della Piattaforma

Public Pressure è una piattaforma web che mette in contatto artisti musicali e collezionisti, consentendo la pubblicazione, l'acquisto e la rivendita di musica sotto forma di NFT. Questa sezione ne illustra le funzionalità principali dal punto di vista dell'utente e descrive a grandi linee come i diversi componenti del sistema collaborano per erogarle.

Cosa può fare l'utente

La piattaforma prevede due tipi principali di attori: gli **artisti** (o le etichette discografiche che li rappresentano) e i **collezionisti**.

Un **artista** può registrarsi sulla piattaforma, creare il proprio profilo pubblico e pubblicare i propri brani come NFT. Per ogni brano è possibile scegliere se coniare una singola edizione unica oppure un numero limitato di copie (denominate *edizioni*); ciascuna edizione è un token distinto e indipendente sulla blockchain. L'artista può fissare un prezzo di vendita diretta, oppure avviare un'asta con la propria base d'asta. Una volta stabilite le condizioni, l'artista può effettuare il *mint*, ovvero la creazione effettiva degli NFT sulla blockchain. Ogni volta che un'edizione viene rivenduta sul mercato secondario, l'artista riceve automaticamente una percentuale sull'importo

della transazione, grazie al meccanismo di royalty implementato nello smart contract (standard ERC-2981).

Un **collezionista** può sfogliare il catalogo della piattaforma, filtrare per artista, genere musicale o tipo di contenuto, e acquistare le edizioni disponibili. L'acquisto può avvenire in tre modi: con carta di credito o debito tramite Stripe, con criptovaluta tramite Coinbase Commerce, oppure pagando direttamente in USDC (un token ERC-20 ancorato al dollaro) dal proprio wallet. Una volta completato l'acquisto, il collezionista diventa il proprietario registrato dell'NFT sulla blockchain e può visualizzare la propria collezione direttamente sul sito. Può inoltre rimettere in vendita le proprie edizioni a prezzo fisso o all'asta, e partecipare alle aste degli altri utenti con offerte progressive.

Entrambi i tipi di utente possono gestire il proprio profilo, consultare lo storico delle transazioni, ricevere notifiche sullo stato degli ordini e, nel caso dei collezionisti, collegare il proprio wallet Ethereum per operare direttamente con la blockchain.

Come interagiscono i componenti

Il funzionamento della piattaforma si basa sull'interazione di tre strati principali: il **backend applicativo**, la **blockchain** ed il **database**. La figura 1.1 mostra uno schema semplificato di questa interazione.

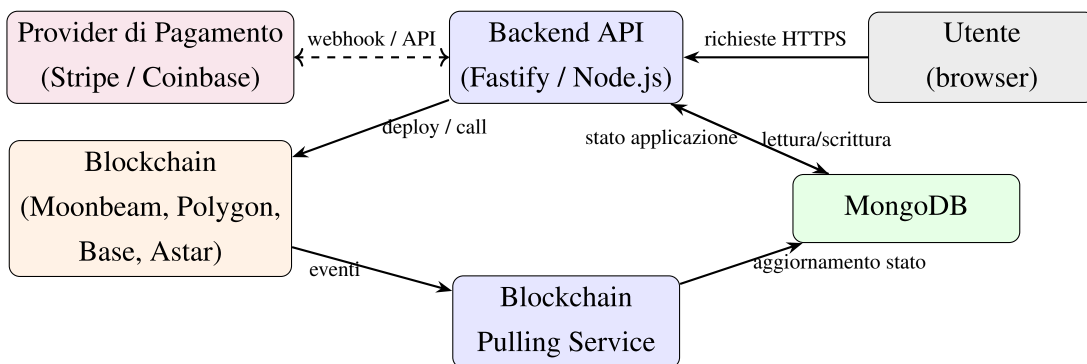

Figura 1.1: Schema ad alto livello dell'interazione tra i componenti di Public Pressure.

Il **backend** è il punto di ingresso per tutte le operazioni degli utenti. Espone un'API REST che gestisce l'autenticazione, le richieste di acquisto, la pubblicazione di nuove opere e tutte le altre operazioni applicative. Ogni richiesta viene validata, autorizzata e tradotta in operazioni sul database o in chiamate verso i sistemi esterni.

Il **database** (MongoDB) conserva lo stato dell'applicazione: profili utente, catalogo dei brani, stato degli ordini, storico delle transazioni e metadati degli NFT. È la fonte di verità per tutto ciò che riguarda il lato applicativo della piattaforma, e viene mantenuto sincronizzato con la blockchain attraverso il servizio di pulling descritto di seguito.

Gli **smart contract** sulla blockchain gestiscono la proprietà degli NFT e le regole economiche della piattaforma: il mint di nuovi token, il trasferimento di proprietà in seguito a un acquisto, la distribuzione automatica delle royalty ai creatori e l'esecuzione delle aste. Quando un utente acquista un NFT, il backend interagisce con lo smart contract per registrare il trasferimento di proprietà in modo permanente e immutabile sulla blockchain.

Il **servizio di blockchain-pulling** è un componente che opera in modo continuativo in background: interroga periodicamente le blockchain supportate alla ricerca di nuovi eventi (trasferimenti, acquisti, offerte) e aggiorna di conseguenza il database dell'applicazione. Questo è necessario perché gli utenti possono interagire con i propri NFT anche al di fuori della piattaforma, ad esempio trasferendoli ad un altro wallet tramite un'applicazione esterna; senza un meccanismo di sincronizzazione, il database potrebbe mostrare informazioni di proprietà obsolete.

I **provider di pagamento** (Stripe per le carte di credito, Coinbase Commerce per le criptovalute) vengono coinvolti solo nella fase di incasso: il backend avvia la sessione di pagamento, attende la conferma tramite webhook e, solo dopo aver ricevuto un esito positivo, procede con il trasferimento dell'NFT. Nella finestra di tempo tra l'avvio e la conferma del pagamento, l'edizione viene temporaneamente riservata per impedire acquisti concorrenti.

1.3 Dichiarazione del Problema

La creazione di un marketplace di NFT musicali multi-blockchain con supporto a pagamenti ibridi presenta numerose sfide tecniche che vanno ben oltre la semplice implementazione di un sito di e-commerce tradizionale. Le principali problematiche affrontate in questo progetto includono:

Gestione della concorrenza nei pagamenti

Gli NFT sono, per definizione, beni unici e a disponibilità limitata. Quando un utente avvia la procedura di acquisto di un NFT, è necessario garantire che lo stesso token non possa essere acquistato contemporaneamente da un altro utente che sta usando un metodo di pagamento diverso. Questa problematica è particolarmente complessa quando si devono gestire simultaneamente pagamenti FIAT (che richiedono conferma da parte di Stripe) e pagamenti in criptovaluta (che richiedono la conferma sulla blockchain).

I tempi di conferma variano significativamente tra i due canali: un pagamento con carta può essere confermato in pochi secondi, mentre una transazione blockchain può

richiedere da pochi secondi a diversi minuti, a seconda della rete e della congestione. Inoltre, i blocchi più recenti di una blockchain non sono immediatamente “confermati”: possono essere soggetti a riorganizzazioni (reorg) che invalidano le transazioni in essi contenute. Questo periodo in cui un blocco esiste ma non è ancora confermato può durare da 2 a 5 minuti a seconda della blockchain ed introduce un’ulteriore complessità nella gestione degli acquisti.

Sincronizzazione tra blockchain e database

I wallet degli utenti non sono limitati o legati alla piattaforma in modo esclusivo ma sono wallet blockchain a sé stanti. Questo permette agli utenti di acquistare, vendere o trasferire token su altre piattaforme oppure interagendo direttamente con la blockchain. È quindi essenziale implementare un sistema che monitori continuamente le blockchain supportate per rilevare e registrare nel database dell’applicazione qualsiasi evento rilevante (ovvero eventi che riguardano wallet degli utenti registrati sulla piattaforma), garantendo la coerenza tra lo stato on-chain e i dati mostrati all’utente.

Interoperabilità multi-blockchain

Il supporto a diverse blockchain richiede la gestione di smart contract che, pur condividendo la maggior parte della logica di base (in quanto tutte le blockchain supportate sono Ethereum-based), presentano differenze nella configurazione, nei costi delle operazioni e nelle specifiche tecniche di ciascuna rete.

Scalabilità e affidabilità

Un marketplace destinato a un pubblico internazionale deve essere in grado di gestire carichi di lavoro variabili e picchi di traffico, specialmente durante eventi come il lancio di nuove collezioni (“drop”) da parte di artisti popolari. L’architettura deve garantire alta disponibilità e tempi di risposta contenuti.

1.4 Scopo e Obiettivi della Tesi

Lo scopo principale di questa tesi è documentare e analizzare la progettazione e l’implementazione del backend e dell’integrazione blockchain del marketplace Public Pressure, con particolare enfasi sulle soluzioni adottate per affrontare le sfide sopra descritte.

Gli obiettivi specifici della tesi sono:

1. Descrivere il contesto teorico relativo alle tecnologie blockchain, agli NFT e ai principali standard utilizzati (ERC-721, ERC-1155, ERC-2981, EIP-2535).
2. Illustrare le tecnologie e gli strumenti selezionati per lo sviluppo del sistema, motivando le scelte effettuate.
3. Presentare l'architettura complessiva del sistema, con particolare attenzione alla struttura orientata a servizi, al design del database e al flusso di comunicazione tra i componenti.
4. Descrivere in dettaglio l'implementazione del backend, inclusa la gestione delle API REST, il sistema di autenticazione, la validazione dei dati e l'integrazione con servizi esterni.
5. Analizzare gli smart contract sviluppati in Solidity, con focus sul Diamond Pattern (EIP-2535), sul meccanismo di asta GBM e sul sistema di gestione delle royalty.
6. Esaminare le soluzioni implementate per la gestione dei pagamenti e della concorrenza tra metodi di pagamento diversi.
7. Descrivere il servizio di sincronizzazione blockchain-database (blockchain-pulling) e le strategie adottate per gestire i blocchi non ancora confermati.
8. Presentare la strategia di testing adottata, inclusi i test di integrazione delle API, i test degli smart contract e l'integrazione nella pipeline CI/CD.
9. Illustrare l'infrastruttura cloud, il processo di deploy e le pratiche DevOps adottate.
10. Discutere le misure di sicurezza implementate a livello applicativo e contrattuale.

1.5 Contesto del Tirocinio

Il lavoro descritto in questa tesi è stato svolto nell'ambito di un tirocinio curriculare della durata di sei mesi (dal 6 marzo 2022 al 12 agosto 2022) presso **Efebia s.r.l.**, azienda con sede legale in Via Nicola Antonio Porpora 63, Milano. Il tirocinio si è svolto interamente in modalità smartworking.

Efebia s.r.l. è un'azienda specializzata nello sviluppo di soluzioni software innovative, con particolare competenza nelle tecnologie blockchain e nelle applicazioni decentralizzate (DApp). L'azienda ha intrapreso lo sviluppo di Public Pressure con l'obiettivo di creare un marketplace di riferimento per gli NFT musicali, distinguendosi dai competitor per il supporto multi-blockchain e per l'attenzione all'esperienza utente.

Il tirocinio, è stato svolto operando all'interno del team di sviluppo backend. Le attività svolte hanno riguardato principalmente:

- Lo sviluppo e la manutenzione delle API REST del backend.
- L'implementazione e il deploy degli smart contract sulle diverse blockchain supportate.
- La progettazione e l'implementazione del servizio di sincronizzazione blockchain-database.
- La gestione dell'integrazione con i sistemi di pagamento (Stripe, Coinbase Commerce).
- La scrittura di test automatizzati e l'integrazione nella pipeline CI/CD.
- Attività di DevOps relative al deploy e alla gestione dell'infrastruttura cloud su AWS.

Come previsto dal progetto formativo, il tirocinio è iniziato con un percorso di formazione iniziale che ha incluso lo studio delle basi teoriche della tecnologia Bitcoin (UTXO, blocchi, gossip protocol, LevelDB), un approfondimento sulle specifiche di Ethereum (modello account-based, Opcodes, smart contract, OpenZeppelin) e un'introduzione all'ecosistema Polkadot (relay chain, parachain, bridge, Substrate).

Il lavoro descritto in questa tesi è opera di un team di sviluppo composto da sei persone. Data la natura ridotta del team, ogni componente del progetto è stato sviluppato attraverso un processo collaborativo, con ciascun membro coinvolto in più aree di lavoro. Il risultato finale è pertanto frutto di un contributo collettivo distribuito, piuttosto che della somma di componenti individuali. Tra le attività svolte, il contributo prevalente ha riguardato l'implementazione del backend, la gestione dei pagamenti ed il testing e la qualità del software, mentre per quanto concerne l'architettura del sistema, lo sviluppo degli smart contract, il deploy su cloud, e le misure di sicurezza il lavoro è stato condiviso in egual misura con gli altri membri del team.

1.6 Struttura della Tesi

La tesi è organizzata nei seguenti capitoli:

Capitolo 2 – Contesto Teorico

Introduce i concetti fondamentali relativi alla tecnologia blockchain, agli NFT e ai principali standard ERC utilizzati nel progetto. Viene inoltre presentata una panoramica dello stato dell'arte dei marketplace di NFT musicali.

Capitolo 3 – Tecnologie e Strumenti
Descrive le tecnologie, i linguaggi di programmazione, i framework e gli strumenti utilizzati nello sviluppo del sistema, motivando le scelte effettuate.

Capitolo 4 – Architettura del Sistema
Presenta il design architetturale complessivo del sistema, la struttura della mono-repo, l'organizzazione dei servizi e il modello dei dati.

Capitolo 5 – Implementazione del Backend
Descrive in dettaglio l'implementazione del backend, incluse le API REST, il sistema di autenticazione, la gestione degli ordini e l'integrazione con i servizi esterni.

Capitolo 6 – Smart Contract e Blockchain
Analizza gli smart contract sviluppati, il Diamond Pattern, il meccanismo di asta GBM, il sistema di royalty e il servizio di sincronizzazione blockchain-database.

Capitolo 7 – Gestione dei Pagamenti
Esamina le soluzioni per la gestione dei pagamenti FIAT e crypto e le strategie per la concorrenza tra metodi di pagamento.

Capitolo 8 – Testing e Qualità del Software
Presenta la strategia di testing, i tipi di test implementati e le metriche di qualità del software.

Capitolo 9 – Deploy e Infrastruttura Cloud
Illustra il processo di deploy, la pipeline CI/CD, l'infrastruttura AWS e le pratiche DevOps adottate.

Capitolo 10 – Sicurezza
Discute le misure di sicurezza implementate a livello applicativo, infrastrutturale e contrattuale.

Capitolo 11 – Conclusioni e Sviluppi Futuri
Riassume i risultati ottenuti e propone possibili evoluzioni del sistema.

Capitolo 2

Contesto Teorico

Questo capitolo introduce i concetti teorici fondamentali necessari per comprendere il lavoro svolto. Vengono descritte le tecnologie blockchain, il concetto di NFT, i principali standard utilizzati e lo stato dell'arte dei marketplace nel settore musicale. Mentre in questo capitolo vengono spiegati i concetti fondamentali della tecnologia blockchain, il capitolo 6 entra più in dettaglio su alcuni concetti centrali per lo sviluppo di questo progetto.

2.1 La Tecnologia Blockchain

Definizione e Principi Fondamentali

Una blockchain è un registro distribuito (distributed ledger) che mantiene una lista di record, chiamati blocchi, collegati tra loro tramite tecniche crittografiche. Ogni blocco contiene un hash crittografico del blocco precedente, un timestamp e i dati delle transazioni. Questa struttura a catena rende estremamente difficile la modifica retroattiva dei dati senza alterare tutti i blocchi successivi, garantendo così l'immutabilità del registro [1].

I principi fondamentali su cui si basa la tecnologia blockchain sono:

- **Decentralizzazione**: non esiste un'autorità centrale che controlla il registro. La rete è composta da nodi peer-to-peer che mantengono ciascuno una copia completa della blockchain.

- **Trasparenza**: tutte le transazioni sono visibili a tutti i partecipanti della rete, garantendo un alto livello di trasparenza.

- **Immutabilità**: una volta che un blocco viene aggiunto alla catena e confermato dalla rete, le transazioni in esso contenute non possono essere modificate o eliminate.
- **Consenso**: i nodi della rete devono raggiungere un accordo (consenso) sullo stato del registro attraverso protocolli specifici (Proof of Work, Proof of Stake, ecc.).

I partecipanti alla rete che mantengono e aggiornano questo registro sono chiamati **nodi**. Un nodo è un qualsiasi computer che esegue il software della blockchain, conserva una copia (completa o parziale) della catena e partecipa alla propagazione delle transazioni. Esistono diverse categorie di nodi con responsabilità diverse:

- **Nodi completi** (*full node*): mantengono una copia integrale di tutta la blockchain a partire dal blocco genesi. Verificano autonomamente ogni transazione e ogni blocco senza fidarsi di terzi.
- **Nodi leggeri** (*light node*): scaricano solo gli header dei blocchi e si affidano ai full node per verificare le transazioni, riducendo i requisiti di storage e banda.
- **Nodi validatori** (*validator node*): nei sistemi Proof of Stake, sono i nodi che hanno depositato una quantità di criptovaluta in garanzia (*stake*) e che hanno il diritto, e la responsabilità, di proporre e certificare nuovi blocchi. Sono loro che eseguono le transazioni sulla EVM, aggiornano lo stato globale e producono i blocchi che vengono aggiunti alla catena.

Fondamenti Crittografici

La blockchain poggia su due primitive crittografiche fondamentali: le funzioni hash e le firme digitali.

Funzioni Hash Crittografiche

Una **funzione hash crittografica** è una funzione che trasforma un input di lunghezza arbitraria in un output di lunghezza fissa (il *digest*), con le seguenti proprietà:

- **Determinismo**: lo stesso input produce sempre lo stesso output.
- **Efficienza**: il calcolo dell'hash è computazionalmente rapido.
- **Resistenza alla preimmagine**: dato un hash h, è computazionalmente troppo costoso trovare un input x tale che $\text{hash}(x) = h$.

- **Effetto valanga**: una piccola variazione dell'input produce un output completamente diverso.
- **Resistenza alle collisioni**: è computazionalmente troppo costoso trovare due input distinti con lo stesso hash.

Bitcoin utilizza l'algoritmo **SHA-256** (output di 256 bit), mentre Ethereum utilizza **Keccak-256** (una variante di SHA-3). Per illustrare l'effetto valanga, si consideri come due stringhe quasi identiche producano hash completamente diversi come dimostrato nel listato 2.1.

```
SHA-256("ciao mondo") =
b94f6f125c79e3a5ffaa826f584c10d52ada669e6762051b826b55776d05a8d

SHA-256("Ciao mondo") =   # solo maiuscola iniziale
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b982
```

Listato 2.1: Esempio di effetto valanga di SHA-256

All'interno della blockchain, ogni blocco contiene l'hash del blocco precedente. Se si tentasse di modificare retroattivamente una transazione nel blocco N, il suo hash cambierebbe, invalidando il riferimento nel blocco $N+1$, e a cascata tutti i blocchi successivi. Per far accettare la modifica alla rete, un attaccante dovrebbe ricalcolare la proof-of-work di tutti i blocchi successivi più velocemente di quanto la rete aggiunga nuovi blocchi — un'impresa computazionalmente irrealizzabile su reti consolidate.

Firme Digitali e Controllo della Proprietà

Ogni partecipante a una blockchain possiede una coppia di chiavi crittografiche asimmetriche (algoritmo **ECDSA** su curva `secp256k1`):

- **Chiave privata**: un numero casuale a 256 bit noto solo al proprietario. Serve per *firmare* le transazioni.
- **Chiave pubblica**: derivata deterministicamente dalla chiave privata, senza rivelare alcuna informazione su di essa.
- **Indirizzo**: su Ethereum, corrisponde agli ultimi 20 byte dell'hash Keccak-256 della chiave pubblica. Rappresenta l'identità pubblica dell'utente sulla rete.

Quando un utente vuole inviare una transazione (ad esempio, trasferire ETH o interagire con uno smart contract), firma i dati della transazione con la propria chiave privata. Chiunque sulla rete può verificare la firma usando la chiave pubblica corrispondente,

accertandosi che: (a) la transazione provenga effettivamente dal titolare della chiave privata, e (b) i dati non siano stati alterati dopo la firma. Questo meccanismo garantisce che nessuno possa spendere fondi o interagire con contratti a nome di un altro utente senza possederne la chiave privata.

Struttura di un Blocco e Albero di Merkle

Un blocco di una blockchain si compone di due parti principali:

- **Header**: contiene i metadati del blocco — hash del blocco precedente, timestamp, nonce (usato nella proof-of-work), e la *Merkle root* delle transazioni.
- **Body**: contiene la lista delle transazioni incluse nel blocco.

La **Merkle root** è la radice di un *albero di Merkle*, una struttura ad albero binario in cui ogni foglia contiene l'hash di una transazione, e ogni nodo interno contiene l'hash della concatenazione dei suoi due figli. La radice riassume in modo univoco tutte le transazioni del blocco.

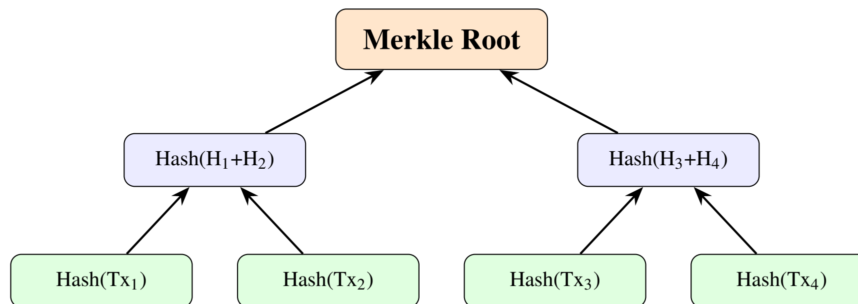

Figura 2.1: Struttura di un albero di Merkle con quattro transazioni. La Merkle root viene inserita nell'header del blocco.

La struttura ad albero di Merkle, visibile nella figura 2.1 ha un'importante proprietà: permette di verificare l'inclusione di una singola transazione nel blocco con una **Merkle proof** in tempo $O(\log n)$, senza dover scaricare tutte le transazioni del blocco. Questo è particolarmente utile per i client leggeri (*light client*) che non mantengono una copia completa della blockchain.

Nell'header di un blocco Ethereum sono presenti in realtà tre radici di Merkle distinte: una per le transazioni, una per lo stato globale (sezione 2.1), ed una per i **receipt**. Un receipt è il documento di esito di una transazione: viene prodotto dal nodo validatore al termine dell'esecuzione e contiene informazioni che non erano note prima di eseguire la transazione, tra cui lo **stato di esecuzione** (successo o revert), il gas effettivamente consumato (discusso nella sezione 2.1), ed i **log** emessi dagli

smart contract. I log sono la rappresentazione on-chain degli eventi Solidity (3.1) (`event`): quando un contratto emette un evento, ad esempio `Transfer(from, to, tokenId)` o `BidPlaced(auctionId, bidder, amount)`, quell'evento viene serializzato ed inserito nel receipt come voce di log, indicizzata per indirizzo del contratto e per topic.

Questa distinzione tra transazione e receipt è fondamentale per chiunque debba monitorare una blockchain dall'esterno: la transazione dice *cosa è stato richiesto*, il receipt dice *cosa è effettivamente successo*. Un'applicazione che vuole sapere se un NFT è stato trasferito non deve analizzare tutta la transazione ma può semplicemente leggere i log nel receipt e cercare un evento `Transfer` con i parametri attesi.

Meccanismi di Consenso

Il consenso distribuito risolve il problema fondamentale: come possono nodi che non si fidano l'uno dell'altro concordare sullo stato del registro? Le due strategie principali adottate dalle blockchain di interesse per questo progetto sono:

Proof of Work (PoW)

Utilizzato da Bitcoin, il PoW richiede che i nodi (*miner*) risolvano un puzzle computazionale: trovare un valore `nonce` tale che l'hash dell'header del blocco sia inferiore a un target prefissato (*difficoltà*). Questo processo richiede un elevato sforzo computazionale (*mining*), ma la verifica della soluzione è immediata.

Il PoW garantisce sicurezza attraverso il costo energetico: per riscrivere la storia della blockchain, un attaccante dovrebbe controllare oltre il 50% della potenza computazionale totale della rete (*attacco 51%*). Su reti consolidate come Bitcoin, questo è economicamente proibitivo. Il principale svantaggio del PoW è l'elevato consumo energetico.

Proof of Stake (PoS)

Ethereum ha abbandonato il PoW nel settembre 2022 (evento noto come *The Merge*) in favore del PoS, che seleziona i validatori in proporzione alla quantità di criptovaluta depositata in garanzia (*stake*). I validatori che si comportano in modo disonesto rischiano la perdita del loro stake (*slashing*).

Il PoS riduce drasticamente il consumo energetico (si stima una riduzione del 99,95% per Ethereum) ed è il meccanismo adottato anche dalle blockchain utilizzate in Public Pressure: Polygon usa una variante di PoS, Moonbeam adotta il Nominated

Proof-of-Stake di Polkadot, e Base eredita la sicurezza da Ethereum tramite il protocollo Optimistic Rollup.

Ethereum e il Modello Account-Based

Ethereum, proposto da Vitalik Buterin nel 2014 [2] e descritto formalmente da Gavin Wood nel Yellow Paper [3], è una piattaforma blockchain progettata per supportare non solo il trasferimento di valore, ma anche l'esecuzione di logica computazionale arbitraria tramite i cosiddetti smart contract. Per gestire lo stato della rete e le transazioni tra i partecipanti, Ethereum adotta un modello account-based, concettualmente vicino al funzionamento di un conto bancario tradizionale. In questo modello, lo stato globale della rete è rappresentato come un insieme di account, ciascuno identificato da un indirizzo a 20 byte derivato crittograficamente (wallet), e associato a un saldo in Ether (la valuta nativa della piattaforma), un contatore di transazioni (nonce), e, nel caso degli account contratto, al codice eseguibile e allo storage persistente del contratto stesso.

Esistono due tipi di account:

- **Externally Owned Account (EOA)**: controllati da chiavi private, possono inviare transazioni e interagire con i contratti.
- **Contract Account**: controllati dal codice del contratto, possono essere attivati solo da transazioni inviate da EOA o da altri contratti. Non dispongono di una chiave privata propria e la logica del contratto viene eseguita automaticamente in risposta alle transazioni ricevute

Ogni transazione inviata da un EOA modifica lo stato globale della rete: può trasferire Ether tra account, invocare funzioni di uno smart contract, o distribuire un nuovo contratto sulla rete. Questo aggiornamento di stato avviene in modo atomico e deterministico attraverso l'Ethereum Virtual Machine (EVM), un ambiente di esecuzione isolato e replicato su tutti i nodi della rete, che garantisce che il medesimo input produca sempre lo stesso output indipendentemente dal nodo che lo elabora.

Smart Contract: Definizione e Ciclo di Vita

Uno **smart contract** è un programma autonomo memorizzato permanentemente sulla blockchain la cui esecuzione è *deterministica*, *trasparente* e *verificabile da qualsiasi nodo della rete*. Il termine fu coniato da Nick Szabo negli anni '90 per descrivere accordi contrattuali la cui esecuzione viene automatizzata senza la necessità di un intermediario di fiducia.

Uno smart contract è scritto in un linguaggio ad alto livello, solitamente Solidity. La compilazione del codice in Solidity genera il **bytecode**, ovvero il codice esadecimale che la EVM può eseguire. La dimensione del bytecode per un singolo contratto non può superare 24KB. Questo limite è stato introdotto con EIP-170 (Ethereum Improvement Proposal) come misura preventiva contro alcuni tipi di attacchi come per esempio la creazione di contratti dal costo di deploy contenuto ma con un costo (in termini di risorse) molto alto, che potevano portare ad un rallentamento della rete in caso di attacco distribuito.

Su Ethereum, gli smart contract vengono eseguiti dalla **Ethereum Virtual Machine (EVM)** e presentano le seguenti caratteristiche essenziali:

- **Immutabilità del codice**: una volta fatto il deploy, il codice non può essere modificato (salvo mediante pattern di proxy come il Diamond Standard approfontito nella sezione 2.2).
- **Esecuzione atomica**: ogni transazione ha esito completamente positivo o viene annullata per intero (*revert*); non esistono stati intermedi persistenti.
- **Assenza di fiducia**: le regole del contratto sono codificate nel bytecode e applicate automaticamente; nessuna delle parti può modificarne il comportamento unilateralmente.
- **Determinismo**: dato lo stesso stato iniziale e la stessa transazione di input, tutti i nodi ottengono il medesimo risultato.

Per illustrare concretamente questi concetti, si consideri lo smart contract nel listato 2.2, scritto in Solidity, che implementa un semplice contatore con controllo di accesso. Questo esempio introduce diversi concetti chiave di Solidity:

- `msg.sender`: l'indirizzo dell'account che ha inviato la transazione corrente. È la principale fonte di informazione sull'identità del chiamante.
- `require`: precondizione che, se non soddisfatta, causa il revert della transazione restituendo il gas non consumato al mittente.
- `event` / `emit`: meccanismo per registrare informazioni nel *transaction log* della blockchain. I log sono consultabili in modo efficiente dall'esterno.
- `view`: modificatore che dichiara che la funzione non altera lo stato; le chiamate `view` non richiedono una transazione on-chain e non hanno costo in gas.

Il ciclo di vita completo di uno smart contract si articola nelle fasi seguenti:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Counter {
  uint256 private count;   // storage permanente sulla blockchain
  address public owner;

  event CountChanged(uint256 newValue, address changedBy);

  constructor() {
    owner = msg.sender;  // chi esegue il deploy diventa owner
    count = 0;
  }

  function increment() external {
    require(msg.sender == owner, "Solo l'owner puo' incrementare");
    count += 1;
    emit CountChanged(count, msg.sender);
  }

  function getCount() external view returns (uint256) {
    return count;  // "view": non modifica lo stato, non consuma gas
  }
}
```

Listato 2.2: Smart contract di esempio: contatore con controllo di accesso

1. **Scrittura**: il contratto viene scritto in un linguaggio di alto livello, tipicamente **Solidity** [25].

2. **Compilazione**: il compilatore `solc` trasforma il codice sorgente in **bytecode EVM** e genera la Application Binary Interface (ABI), un descrittore JSON che specifica le funzioni, i parametri e gli eventi del contratto, necessario per interagirvi dall'esterno.

3. **Deploy**: una transazione speciale, priva di destinatario, contiene il bytecode di creazione. L'EVM esegue il costruttore e assegna un indirizzo permanente al contratto.

4. **Interazione**: gli utenti inviano transazioni all'indirizzo del contratto, codificando la chiamata di funzione secondo l'ABI. I primi 4 byte del payload sono il *function selector*: i 4 byte iniziali dell'hash Keccak-256 della firma della funzione (es. `keccak256("increment()")` $\rightarrow$ `0xd09de08a`).

5. **Aggiornamento** (opzionale): tramite pattern come il Diamond Standard (Sezione 2.2), è possibile modificare la logica mantenendo l'indirizzo del contratto invariato.

Modello di Esecuzione: Contratti come Bytecode Passivo

Un aspetto fondamentale degli smart contract è che **un contratto non è un processo in esecuzione continua**. Non esiste nessun server, nessun thread, nessun daemon che rimane in ascolto in attesa di richieste. Un contratto è semplicemente **bytecode memorizzato sulla blockchain**, associato a un indirizzo e a uno spazio di storage persistente. Tra una transazione e l'altra il contratto è inerte: non consuma risorse, non può auto-attivarsi, non può fare nulla da solo.

L'esecuzione avviene esclusivamente su richiesta esplicita: quando una transazione viene inviata all'indirizzo del contratto, il nodo validatore che elabora il blocco istanzia temporaneamente la EVM, carica il bytecode del contratto, lo esegue fino al termine (o fino a un `revert`), aggiorna lo storage e produce un ricevuta. Terminata l'esecuzione, non rimane nessun processo attivo.

Questa natura passiva ha una conseguenza diretta sul modello di concorrenza: **l'EVM non può eseguire transazioni in parallelo**. Ogni blocco contiene un insieme ordinato di transazioni che vengono processate una alla volta, in sequenza. Le transazioni inviate contemporaneamente da più utenti non si eseguono in parallelo: convergono in una coda (*mempool*), vengono incluse in un blocco con un ordine preciso, e vengono eseguite in quell'ordine. Non esistono quindi race condition sullo storage del contratto,

ed il concetto di “sovraccarico” di un contratto non ha significato. Le transazioni in eccesso restano nel mempool e vengono incluse nei blocchi successivi. Nella Figura 2.2 viene esemplificata la mempool contenente transazioni in attesa di essere processate ed aggiunte in un blocco.

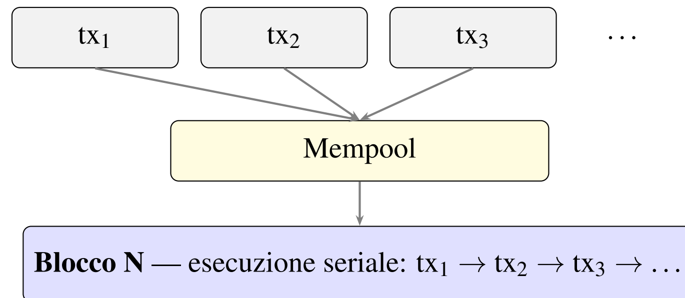

Figura 2.2: Le transazioni convergono nel mempool e vengono incluse nei blocchi in ordine seriale. Non esiste esecuzione parallela nell’EVM.

Gas e Costi di Esecuzione

Il **gas** è l’unità di misura del lavoro computazionale necessario per eseguire operazioni sull’EVM. Ogni operazione (opcode) ha un costo in gas predeterminato; il mittente di una transazione specifica un `gasLimit` (massimo gas spendibile), che rappresenta il tetto massimo di gas che la transazione è autorizzata a consumare. Il prezzo del gas è determinato da due componenti: una `baseFee`, espressa in Gwei ($1\,\text{Gwei} = 10^{-9}\,\text{ETH}$), calcolata automaticamente dal protocollo in funzione della congestione della rete e interamente bruciata, e una `priorityFee` opzionale che il mittente può aggiungere come incentivo per i validatori a dare priorità alla transazione in modo che sia processata prima. Il costo effettivo per unità di gas è quindi la somma di queste due componenti. Il costo effettivo di una transazione è:

$$\text{Costo (ETH)} = \text{gasUsed} \times \text{gasPrice}$$

La Tabella 2.1 riporta il costo di alcuni opcode fondamentali per dare un’idea dell’ordine di grandezza:

Tabella 2.1: Costo in gas di operazioni EVM comuni (valori post-EIP-3529)

Opcode	**Operazione**	**Gas**
`ADD`, `MUL`	Aritmetica di base	3–5
`SHA3`	Hash Keccak-256	30 + 6/word
`MLOAD` / `MSTORE`	Lettura/scrittura in memoria	3
`SLOAD`	Lettura da storage	2100
`SSTORE` (nuovo slot)	Scrittura in storage	20000
`CALL`	Chiamata a contratto esterno	700+
`CREATE`	Deploy di un nuovo contratto	32000

La forte differenza di costo tra memoria (volatile, cancellata al termine dell'esecuzione) e storage (persistente sulla blockchain) ha importanti implicazioni architetturali: gli smart contract ottimizzati minimizzano le scritture in storage. Il meccanismo del gas impedisce loop infiniti e incentiva alla scrittura di codice ottimizzato: se il gas si esaurisce durante l'esecuzione, la transazione viene annullata (*out of gas*) e le modifiche di stato vengono ripristinate.

Aree di Memoria nell'EVM

L'EVM distingue tre aree di dati con semantiche molto diverse:

- **Storage**: persistente tra le transazioni, ogni contratto ha il proprio spazio di storage. È il livello più costoso (20.000 gas per la prima scrittura di un nuovo valore). In Solidity le variabili di stato risiedono qui.
- **Memory**: volatile, allocata a ogni chiamata di funzione e cancellata al termine. Usata per array temporanei e valori intermedi. Ha costo basso (3 gas per parola) ma cresce quadraticamente oltre i primi 724 byte.
- **Calldata**: area di sola lettura che contiene i dati della transazione (parametri della funzione). Usare `calldata` invece di `memory` per i parametri di funzioni `external` è una comune ottimizzazione del gas.

Vale la pena chiarire dove lo storage risiede fisicamente, poiché questo ha implicazioni dirette su come funziona il Diamond Pattern. Ethereum mantiene uno **stato globale** (world state): una struttura dati che associa a ogni indirizzo della rete il suo saldo in Ether, il suo bytecode (se è un contratto) ed il suo storage. Questo stato è organizzato come un albero di Merkle ed è conservato nel database sul disco di ogni nodo validatore della rete. Lo storage di un contratto non è quindi RAM occupata

da un processo in esecuzione, ma una porzione del database del nodo, identificata dall'indirizzo del contratto e persistente tra una transazione e l'altra.

Quando la EVM esegue una transazione, carica i dati necessari dal database, esegue le istruzioni in RAM come calcolo temporaneo, poi scrive le modifiche nel database. Al termine dell'esecuzione non rimane nulla in memoria: tutto lo stato persistente torna sul disco, pronto per la prossima transazione.

Opcodes e EVM

L'Ethereum Virtual Machine è una **macchina stack-based**: ogni opcode legge i propri operandi dalla cima dello stack e vi deposita i risultati. Lo stack ha una profondità massima di 1024 elementi, ciascuno da 256 bit. Le operazioni più rilevanti per lo sviluppo di smart contract avanzati includono:

- `PUSH1`–`PUSH32`: carica una costante sullo stack.
- `ADD`, `SUB`, `MUL`, `DIV`: aritmetica a 256 bit.
- `SLOAD` / `SSTORE`: legge/scrive dal persistent storage.
- `CALL` / `DELEGATECALL` / `STATICCALL`: chiamate a contratti esterni con differenti semantiche di contesto.
- `REVERT` / `RETURN`: termina l'esecuzione, con o senza annullamento delle modifiche.

La comprensione degli opcode è fondamentale per l'ottimizzazione del gas e per la scrittura di assembly inline in Solidity, come nel caso del meccanismo `fallback` del Diamond Pattern.

Blockchain Layer 2 e soluzioni di scalabilità

Le blockchain Layer 2 sono protocolli costruiti sopra una blockchain principale (Layer 1) con l'obiettivo di aumentarne la capacità di elaborazione delle transazioni, riducendo contestualmente i costi ed i tempi di attesa, senza alterare il protocollo base sottostante. Il problema che queste soluzioni si propongono di risolvere è intrinseco all'architettura delle blockchain di prima generazione: garantire sicurezza e decentralizzazione ha un costo in termini di throughput, e reti come Ethereum sono in grado di processare solo un numero limitato di transazioni al secondo. Nei periodi di alta domanda, questo si traduce in commissioni elevate e congestione della rete.

Le soluzioni Layer 2 nascono proprio per alleggerire questo carico, spostando l'esecuzione delle transazioni fuori dalla catena principale e pubblicando sul Layer 1 solo un riassunto crittografico del risultato, riducendo drasticamente il numero di operazioni che devono essere elaborate direttamente sulla rete principale. Un aspetto fondamentale di questo approccio è che il Layer 2 non deve costruirsi una propria rete di validatori indipendente, ma eredita la sicurezza del Layer 1 sottostante, mantenendo le garanzie crittografiche di quest'ultimo.

La famiglia di soluzioni Layer 2 più diffusa oggi è quella dei Rollup, che raggruppa molte transazioni in lotti (batch) e ne pubblica una prova aggregata sul Layer 1. Esistono due varianti principali: gli Optimistic Rollup [28], che assumono la validità delle transazioni e prevedono un periodo di contestazione durante il quale eventuali frodi possono essere segnalate, e gli ZK-Rollup [29], che utilizzano prove crittografiche a conoscenza zero (zero-knowledge proofs [27]) per garantire matematicamente la correttezza di ogni batch senza necessità di un periodo di attesa. Tra i casi più rappresentativi di queste tecnologie vi sono Base e Polygon, blockchain Layer 2 costruite sull'ecosistema Ethereum con approcci e obiettivi parzialmente distinti.

Base è un Optimistic Rollup sviluppato da Coinbase, uno dei maggiori exchange di criptovalute al mondo, lanciato nel 2023. È costruito utilizzando l'OP Stack, il framework open-source alla base di Optimism, e si distingue per la forte integrazione con l'ecosistema Coinbase, che ne facilita l'adozione da parte di utenti già familiari con quella piattaforma. Base non dispone di un proprio token nativo (le commissioni vengono pagate in ETH) e si posiziona esplicitamente come soluzione orientata all'adozione di massa, offrendo commissioni molto contenute rispetto al Layer 1 e un'esperienza utente semplificata.

Polygon, invece, propone un approccio più articolato alla scalabilità, sviluppando Polygon zkEVM: un Layer 2 basato su ZK-Rollup che utilizza prove a conoscenza zero per garantire la correttezza delle transazioni, mantenendo al contempo la piena compatibilità con l'Ethereum Virtual Machine. Polygon dispone di un proprio token nativo, POL, utilizzato per il pagamento delle commissioni e per lo staking dei validatori, e si rivolge prevalentemente a sviluppatori e realtà enterprise che necessitano di scalabilità elevata e flessibilità architetturale.

Ecosistema Polkadot e Blockchain Ethereum-Compatibili

Polkadot, progettato da Gavin Wood (co-fondatore di Ethereum) [22], introduce un'architettura multi-chain composta da:

- **Relay Chain**: la blockchain principale che gestisce il consenso, la sicurezza condivisa e la comunicazione inter-chain.
- **Parachain**: blockchain specializzate che si collegano alla Relay Chain, beneficiando della sua sicurezza. Ogni parachain può avere la propria logica e le proprie regole. Può ad esempio emettere un proprio token nativo, con politiche di inflazione, commissioni di transazione e meccanismi di staking completamente indipendenti da quelli della relay chain. Una parachain può supportare un ambiente di esecuzione completamente diverso dall'EVM, come per esempio **Astar** che supporta WebAssembly (Wasm) oltre ad EVM.
- **Bridge**: componenti che permettono la comunicazione tra Polkadot e blockchain esterne come Ethereum e Bitcoin.
- **Substrate**: un framework modulare per la costruzione di blockchain personalizzate, utilizzato per creare le parachain di Polkadot.

Moonbeam è una parachain dell'ecosistema Polkadot che fornisce piena compatibilità con l'EVM di Ethereum. Questo significa che gli smart contract scritti in Solidity per Ethereum possono essere distribuiti su Moonbeam senza modifiche, beneficiando al contempo dell'interoperabilità offerta dall'ecosistema Polkadot. Moonbeam è stata una delle prime blockchain supportate da Public Pressure.

Analogamente, altre blockchain supportate dal progetto come **Polygon** (Layer 2 di Ethereum), **Base** (Layer 2 basato su Optimism) e **Astar** (parachain di Polkadot) sono tutte Ethereum-compatibili, il che ha reso possibile l'utilizzo degli stessi smart contract su reti diverse con modifiche minime alla configurazione.

Conferma dei Blocchi e Riorganizzazioni

In una blockchain, il tempo di conferma di un blocco rappresenta l'intervallo necessario affinché un blocco possa essere considerato permanente e non più soggetto a revisione. Quando un nuovo blocco viene aggiunto alla catena, esso non è immediatamente definitivo: in presenza di più nodi che producono blocchi in modo concorrente, possono emergere temporaneamente versioni alternative della catena (fork), nelle quali blocchi diversi competono per occupare la stessa posizione.

Il protocollo di consenso risolve questa ambiguità selezionando la catena valida, con la conseguenza che un blocco non ancora confermato potrebbe essere scartato a favore di un blocco concorrente, rendendo di fatto nulle le transazioni in esso contenute. La conferma è quindi la garanzia crittografica ed economica che, superata una certa soglia

di blocchi successivi ad un blocco di interesse, questo non possa più essere rimosso o sostituito dalla catena senza un costo proibitivo per chi tentasse di farlo.

I meccanismi di risoluzione dei fork variano da protocollo a protocollo, ma convergono tutti verso il medesimo obiettivo: garantire che la rete raggiunga una visione condivisa e univoca dello stato della catena. Per fare alcuni esempi esiste **Longest Chain** che da priorità al ramo più lungo, **GHOST** che prende in considerazione un peso oltre che la lunghezza del ramo ed altre varianti, quasi tutte derivate in modo diretto o indiretto da Longest Chain.

Per questo motivo, si considerano "confermate" solo le transazioni incluse in blocchi sufficientemente profondi nella catena. Il numero di conferme necessarie varia in base alla blockchain:

- Ethereum: tipicamente 12-15 blocchi (circa 3 minuti).
- Polygon: 128 blocchi (circa 5 minuti).
- Moonbeam: 4-5 blocchi, (circa 30 secondi).

Questa caratteristica ha importanti implicazioni per il servizio di blockchain-pulling di Public Pressure, come verrà discusso nel Capitolo 6.

2.2 Non-Fungible Token (NFT)

Definizione e Caratteristiche

Un Non-Fungible Token (NFT) è un token digitale registrato su una blockchain che rappresenta un bene unico e non intercambiabile. A differenza delle criptovalute come Bitcoin o Ether, che sono **fungibili** (ogni unità è identica ed intercambiabile con un'altra), ogni NFT ha un identificatore unico e può avere attributi e metadati distinti [4].

La distinzione tra fungibile e non fungibile è analoga a quella tra una banconota da 10 euro (fungibile: una vale quanto un'altra) e un'opera d'arte originale (non fungibile: ogni pezzo è unico). Un NFT musicale può rappresentare, ad esempio, la proprietà di una traccia audio in edizione limitata: chi detiene il token è il proprietario certificato sulla blockchain di quell'edizione specifica. Questo paragone non è perfetto in quanto possono esistere più NFT identici fra loro (dette edizioni), però, anche se i metadati sono identici, ogni singola edizione è un token distinto ed unico sulla blockchain.

Quando si parla di creazione di un NFT si usa il termine minting, letteralmente coniare. Il termine venne usato per le criptomonete ed è rimasto in uso anche per i token non fungibili.

Le caratteristiche fondamentali degli NFT sono:

- **Unicità**: ogni NFT ha un identificatore unico all'interno del suo contratto, che lo distingue da tutti gli altri token.
- **Indivisibilità**: un NFT non può essere suddiviso in frazioni.
- **Provenienza verificabile**: la blockchain registra l'intera storia di un NFT, dalla creazione (minting) a ogni trasferimento di proprietà successivo.
- **Proprietà dimostrabile**: il possessore di un NFT può dimostrare crittograficamente la propria proprietà tramite la funzione `ownerOf(tokenId)`, una delle funzioni definite dallo standard ERC-721, come è possibile vedere nella tabella 2.3.
- **Interoperabilità**: grazie agli standard condivisi, gli NFT possono essere visualizzati, scambiati e interagire con diverse piattaforme e applicazioni. Quindi operazioni eseguite al di fuori di Public Pressure saranno visibili all'interno della piattaforma e viceversa.

Standard ERC-721

Lo standard ERC-721 [4] è stato il primo standard per i NFT su Ethereum, proposto nel 2018. Definisce un'interfaccia minima che un contratto deve implementare per gestire token non fungibili.

Il listato 2.3 mostra l'interfaccia ERC-721 e contiene la lista delle funzioni che è necessario implementare per poter creare ed interagire con NFT rispettando lo standard. La funzione `ownerOf(tokenId)` è il cuore dello standard: dato l'identificatore numerico di un token, restituisce l'indirizzo Ethereum del suo proprietario corrente. L'evento `Transfer`, il quale viene usato per trasferire un token da un certo wallet ad un altro e di conseguenza viene emesso a ogni cambio di proprietà, permette di ricostruire l'intera storia di un token a partire dai log della blockchain senza dover consultare un database centralizzato.

In Public Pressure, il contratto `ERC721Multiple` implementa questo standard attraverso le librerie OpenZeppelin Upgradeable [8], aggiungendo funzionalità come l'enumerazione dei token (`ERC721Enumerable`), la possibilità di mettere in pausa i trasferimenti (`Pausable`) e il supporto all'aggiornabilità del contratto (`UUPSUpgradeable`).

```
interface IERC721 {
  event Transfer(address indexed from, address indexed to,
  uint256 indexed tokenId);
  event Approval(address indexed owner, address indexed
 approved,
  uint256 indexed tokenId);

  function balanceOf(address owner) external view
  returns (uint256 balance);
  function ownerOf(uint256 tokenId) external view
  returns (address owner);
  function safeTransferFrom(address from, address to,
  uint256 tokenId) external;
  function approve(address to, uint256 tokenId) external;
  function getApproved(uint256 tokenId) external view
  returns (address operator);
  function setApprovalForAll(address operator, bool approved)
  external;
}
```

Listato 2.3: Interfaccia principale ERC-721

Standard ERC-1155

Lo standard ERC-1155 [5] introduce il concetto di **multi-token**: un singolo contratto può gestire sia token fungibili che non fungibili. Questo è particolarmente utile nel contesto di Public Pressure, dove un artista potrebbe voler creare più edizioni dello stesso brano musicale (ad esempio, 100 copie di un brano in edizione limitata).

A differenza di ERC-721, dove ogni token ha un ID univoco, in ERC-1155 ogni ID rappresenta un *tipo* di token, e per ogni tipo possono esistere più copie. Questo riduce significativamente i costi di gas per il minting di più token simili in quanto questo può essere eseguito una sola volta per creare multiple edizioni e di conseguenza multipli NFT con una singola azione.

Standard ERC-2981: NFT Royalty

Lo standard ERC-2981 [6] definisce un'interfaccia standard per il pagamento delle royalty sulle vendite secondarie degli NFT. Quando un NFT viene rivenduto sul mercato secondario, il contratto può indicare a chi e quanto deve essere pagato come royalty, come si può notare nel listato 2.4. Questa funzione restituisce l'indirizzo del destinatario delle royalty e l'importo dovuto in base al prezzo di vendita. In Public Pressure, le royalty vengono distribuite automaticamente sia nelle vendite all'asta (GBM) che nelle vendite a prezzo fisso (TokenSale), garantendo che gli artisti ricevano una percentuale su ogni rivendita dei loro NFT.

```
interface IERC2981 {
  function royaltyInfo(uint256 tokenId, uint256 salePrice)
  external view returns (
  address receiver,
  uint256 royaltyAmount
  );
}
```

Listato 2.4: Interfaccia ERC-2981 per le royalty

EIP-2535: Diamond Standard

Lo standard EIP-2535 [7], noto anche come **Diamond Standard**, è un pattern di proxy che permette a un singolo smart contract di esporre funzionalità implementate in contratti separati chiamati **facet**. La nomenclatura fa riferimento ad un diamante e le sue sfaccettature. Il numero di facet di un diamond contract è tecnicamente infinito, limitato soltanto dai costi di deployment ed esecuzione. I vantaggi di questo sistema sono un modo modulare e meno costoso di aggiornare lo smart contract, in quanto se viene trovato un problema, è sufficiente aggiornare soltanto il facet che lo contiene, senza bisogno di aggiornare gli altri facet ed il diamond.

Il meccanismo centrale è l'opcode **DELEGATECALL**: quando il contratto Diamond riceve una chiamata, identifica quale facet contiene la funzione richiesta (tramite il *function selector*) e le delega l'esecuzione. La peculiarità di `DELEGATECALL` rispetto alla normale `CALL` è che il codice del facet viene eseguito nel **contesto di storage del Diamond**: il facet legge e scrive la memoria del proxy, non la propria. Questo garantisce che tutti i facet condividano un unico stato persistente.

L'architettura Diamond si compone di:

- **Diamond (Proxy)**: il contratto principale che riceve tutte le chiamate. Utilizza una funzione `fallback()` con `delegatecall` per inoltrare le chiamate al facet appropriato in base al selettore della funzione.
- **Facet**: contratti che implementano le funzionalità effettive. Ogni facet può contenere una o più funzioni. I facet possono essere aggiunti, sostituiti o rimossi tramite un'operazione chiamata "diamond cut". Ogni Diamond Contract, per rispettare lo standard EIP-2535, deve avere almeno 3 Facet contracts: **DiamondCutFacet**, **DiamondLoupeFacet** ed **OwnershipFacet**.
- **DiamondCutFacet**: il facet che gestisce le operazioni di aggiornamento, permettendo di aggiungere, sostituire o rimuovere funzioni (ovvero aggiungere, sostituire o rimuovere facet).

- **DiamondLoupeFacet**: il facet di introspezione che permette di interrogare il diamond per conoscere i facet e le funzioni disponibili.
- **OwnershipFacet**: il facet che contiene le funzioni riguardanti il proprietario del contratto (un wallet), permette di sapere il proprietario attuale oppure eseguire un cambio del proprietario.
- **AppStorage**: un pattern di storage condiviso che utilizza slot di storage con nome per evitare collisioni tra i diversi facet.

Questo standard è stato scelto per Public Pressure perché permette di aggiornare la logica dei contratti (ad esempio, il meccanismo di asta o la gestione delle vendite) senza dover ridistribuire un nuovo contratto e migrare tutti gli NFT esistenti. La Figura 2.3 mostra l'architettura del Diamond Pattern.

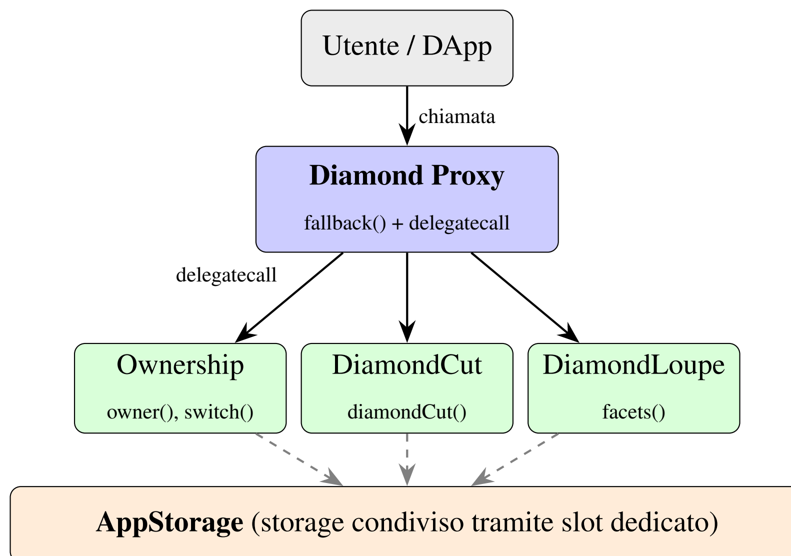

Figura 2.3: Architettura del Diamond Pattern (EIP-2535): il proxy inoltra le chiamate ai facet appropriati tramite delegatecall, condividendo lo storage.

Il Diamond Standard verrà discusso più in dettaglio nel capitolo 6, sezione 6.2.

2.3 NFT nel Settore Musicale

Opportunità per gli Artisti

Gli NFT offrono agli artisti musicali diverse opportunità che i modelli di distribuzione tradizionali non possono fornire:

- **Monetizzazione diretta**: gli artisti possono vendere la propria musica direttamente ai fan senza intermediari, trattenendo una percentuale significativamente superiore rispetto alle piattaforme di streaming.
- **Royalty automatiche**: grazie allo standard ERC-2981, ogni rivendita di un NFT musicale genera automaticamente una royalty per l'artista, creando un flusso di entrate passivo e perpetuo.
- **Edizioni limitate**: gli artisti possono creare edizioni limitate dei propri brani o album, conferendo loro un valore di scarsità e collezionabilità.
- **Contenuti esclusivi**: gli NFT possono essere associati a contenuti esclusivi, esperienze VIP, biglietti per concerti o merchandise fisico (redeemables).
- **Relazione diretta con i fan**: gli NFT creano un legame diretto e verificabile tra artista e collezionista, senza l'intermediazione di piattaforme centralizzate.

Stato dell'Arte dei Marketplace di NFT Musicali

Al momento dello sviluppo di Public Pressure, il panorama dei marketplace di NFT musicali comprendeva diverse piattaforme:

- **Audius**: piattaforma di streaming musicale decentralizzata che ha integrato funzionalità NFT.
- **Catalog**: marketplace specializzato in NFT musicali one-of-one (edizioni singole) su Ethereum.
- **Sound.xyz**: piattaforma per il rilascio di musica come NFT con meccanismo di "listening party" e edizioni limitate.
- **Royal**: marketplace che permette di acquistare quote di royalty musicali sotto forma di NFT, e di diventare quindi proprietari parziali dei diritti d'autore di un brano.
- **OpenSea**: marketplace generalista di NFT che include anche contenuti musicali, ma senza funzionalità specifiche per il settore.

Nessuna di queste piattaforme, al momento dello sviluppo, offriva contemporaneamente il supporto multi-blockchain e i pagamenti ibridi FIAT/crypto che caratterizzano Public Pressure. Questa combinazione di funzionalità rappresenta il principale elemento di differenziazione del progetto.

2.4 Meccanismo di Asta GBM

Il GBM (Gamified Bidding Mechanism) è un meccanismo di asta innovativo in cui i partecipanti che vengono superati da un'offerta più alta ricevono un **incentivo** economico, oltre al rimborso completo della propria offerta. Questo crea un sistema in cui anche i perdenti di un'asta beneficiano dalla partecipazione, incentivando un maggior numero di offerte e, di conseguenza, un prezzo finale più elevato per il venditore.

Il funzionamento del GBM si basa sui seguenti principi:

1. Quando un offerente viene superato, riceve il rimborso della propria offerta più un **incentivo** calcolato in base alla differenza tra la nuova offerta e quella precedente. L'incentivo è limitato da un valore minimo (`incMin`) e un valore massimo (`incMax`) per evitare abusi. L'incentivo cresce proporzionalmente all'eccesso dell'offerta rispetto all'incremento minimo richiesto, moltiplicato per un fattore (`bidMultiplier`).

2. Il meccanismo **Hammer Time** estende automaticamente la durata dell'asta se un'offerta viene piazzata negli ultimi istanti, prevenendo il "sniping" (strategia di piazzare un'offerta all'ultimo secondo).

3. Esiste un **periodo di grazia** dopo la fine dell'asta durante il quale il venditore può annullare la vendita, pagando un costo pari agli incentivi dovuti.

La Figura 2.4 illustra il flusso di un'asta GBM con il meccanismo di incentivi. Questo meccanismo è stato implementato come facet del Diamond contract, come verrà descritto in dettaglio nel Capitolo 6.

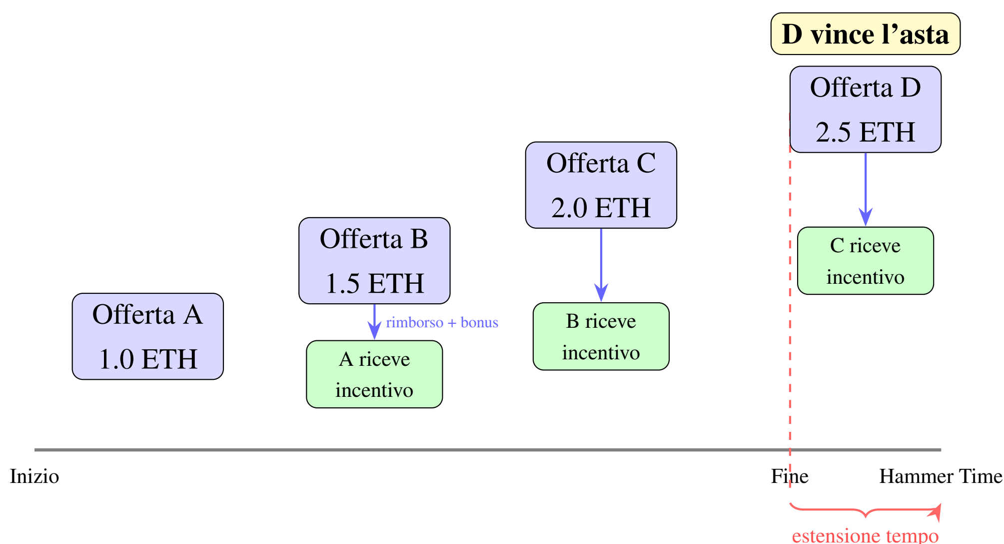

Figura 2.4: Flusso di un'asta GBM: ogni offerente superato riceve un incentivo. Il meccanismo Hammer Time estende la durata se arrivano offerte negli ultimi istanti.

Capitolo 3

Tecnologie e Strumenti Utilizzati

Questo capitolo presenta le tecnologie, i linguaggi di programmazione, i framework e gli strumenti utilizzati nello sviluppo di Public Pressure, spiegando le scelte effettuate in base ai requisiti del progetto. La Figura 3.1 offre una visione d'insieme dello stack tecnologico adottato.

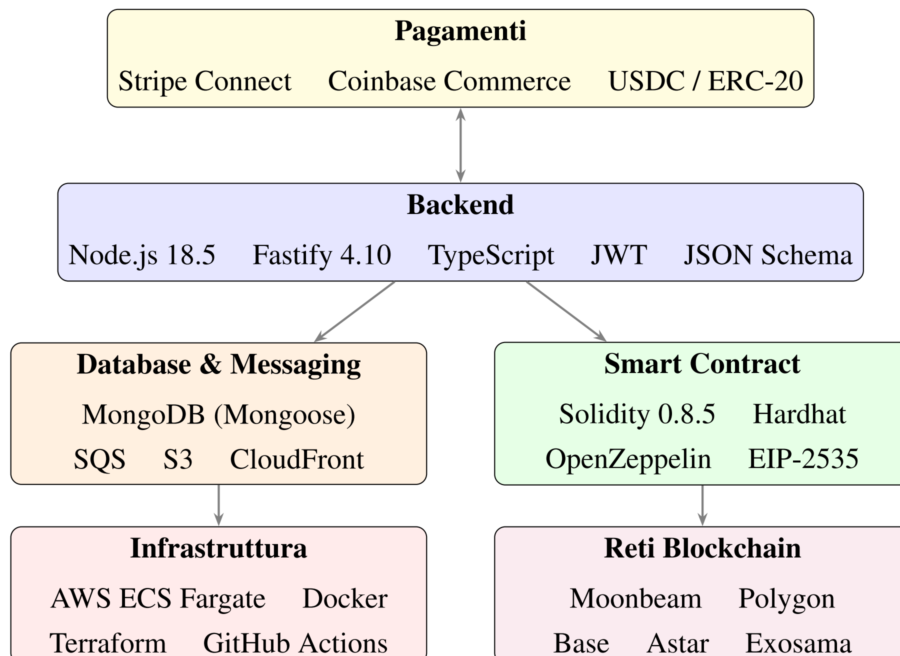

Figura 3.1: Stack tecnologico di Public Pressure: al centro il backend, con le sue dipendenze verso database, smart contract e provider di pagamento.

3.1 Linguaggi di Programmazione

TypeScript e JavaScript

Il backend di Public Pressure è sviluppato in **TypeScript**, un superset tipizzato di JavaScript sviluppato da Microsoft. TypeScript aggiunge a JavaScript un sistema di tipi statico opzionale che consente di individuare errori in fase di compilazione anziché a runtime, migliorando la manutenibilità e la robustezza del codice in progetti di grandi dimensioni.

La configurazione TypeScript del progetto utilizza il target `ESNext` e il sistema di moduli `ESNext`, abilitando le funzionalità più recenti del linguaggio. La compilazione produce codice JavaScript che viene eseguito dall'ambiente di runtime Node.js.

La scelta di TypeScript è stata motivata dalla dimensione e complessità del progetto: una monorepo con 17 pacchetti interconnessi richiede un livello di affidabilità e documentazione del codice che JavaScript semplice non è in grado di garantire.

Solidity

Solidity [25] è il linguaggio di programmazione utilizzato per scrivere gli smart contract. È un linguaggio tipizzato staticamente, progettato specificamente per l'Ethereum Virtual Machine (EVM), e supporta ereditarietà, librerie e tipi complessi definiti dall'utente.

Il progetto utilizza Solidity nella versione `0.8.5`, con l'ottimizzatore del compilatore abilitato a 1000 esecuzioni. Questa configurazione rappresenta un compromesso tra la dimensione del bytecode generato e l'efficienza a runtime: un numero di esecuzioni elevato ottimizza il codice per l'uso frequente, riducendo i costi di gas nelle operazioni più comuni.

3.2 Framework e Runtime

Node.js

Node.js è l'ambiente di runtime JavaScript lato server basato sul motore V8 di Google Chrome. La sua architettura a singolo thread con I/O non bloccante (event loop) lo rende particolarmente adatto per applicazioni che gestiscono un alto numero di connessioni concorrenti con operazioni prevalentemente di I/O, come nel caso di un backend che deve comunicare con code di messaggi e blockchain.

Node.js nella versione `18.5.0` è stato scelto come runtime per tutti i servizi backend del progetto per la sua maturità, l'ampio ecosistema di pacchetti npm e la familiarità del team di sviluppo.

Fastify

Fastify [16] è un framework web per Node.js progettato per offrire prestazioni elevate con un overhead minimo. A differenza di Express.js (il framework Node.js più diffuso), Fastify utilizza un sistema di serializzazione JSON basato su schema che risulta significativamente più veloce nella generazione delle risposte HTTP.

La versione utilizzata nel progetto è `Fastify 4.10.2`. Le caratteristiche di Fastify che lo rendono particolarmente adatto a Public Pressure sono:

- **Sistema a plugin**: Fastify si basa su un'architettura a plugin incapsulati che facilita la separazione delle responsabilità e la modularità del codice. Ogni modulo API è registrato come plugin indipendente.
- **Validazione basata su JSON Schema**: ogni route può definire uno schema JSON per la validazione automatica dei parametri di input (querystring, body, headers) e della risposta. Fastify utilizza AJV (Another JSON Validator) per la compilazione e la validazione degli schemi, eliminando la necessità di validazione manuale.
- **Lifecycle hooks**: Fastify fornisce un sistema di hook che permette di eseguire codice in diverse fasi del ciclo di vita della richiesta (`onRequest`, `preHandler`, `onResponse`, ecc.), utilizzati nel progetto per autenticazione, logging e gestione degli errori.
- **Supporto nativo a OpenAPI**: tramite il plugin `@fastify/swagger`, è possibile generare automaticamente la documentazione OpenAPI 3.1.0 delle API a partire dagli schemi definiti nelle route.

Hardhat

Hardhat [20] è un ambiente di sviluppo per Ethereum che fornisce strumenti per la compilazione, il testing, il deploy ed il debug degli smart contract Solidity. Hardhat è stato scelto rispetto a alternative come Truffle per la sua velocità di compilazione, il sistema di task estensibile e il supporto nativo ai plugin OpenZeppelin per contratti aggiornabili.

Alcune delle funzionalità principali:

- **Hardhat Network** Una rete Ethereum simulata localmente. Permette di sviluppare e testare l'implementazione degli smart contract senza costi di gas.
- **Mainnet Forking** Copia lo stato della rete principale per testare l'implementazione in modo realistico

3.3 Database

MongoDB

MongoDB [17] è un database NoSQL (non relazionale) orientato ai documenti che memorizza i dati in formato BSON (Binary JSON). La scelta di MongoDB come database principale per Public Pressure è motivata da diversi fattori:

- **Flessibilità dello schema**: i modelli di dati di un marketplace NFT sono intrinsecamente variabili. Un template NFT può avere attributi molto diversi a seconda del tipo di media (musica, video, collectible), del tipo di vendita (asta, prezzo fisso, airdrop) e delle opzioni di configurazione. MongoDB consente di gestire questa variabilità senza le rigide restrizioni di uno schema relazionale.
- **Relazioni limitate**: sebbene esistano relazioni tra le entità (un ordine referenzia un template, un artista ha più template, ecc.), queste relazioni sono relativamente semplici e non richiedono operazioni di JOIN complesse. MongoDB supporta le reference tra documenti tramite `ObjectId` e il meccanismo di `populate` di Mongoose.
- **Prestazioni in lettura**: MongoDB eccelle nelle operazioni di lettura, particolarmente importanti per un marketplace dove le query di ricerca e visualizzazione sono molto più frequenti delle operazioni di scrittura.
- **Transazioni multi-documento**: dalla versione 4.0, MongoDB supporta le transazioni ACID multi-documento all'interno di un replica set. Questa funzionalità è cruciale per Public Pressure, dove operazioni come la creazione di un ordine richiedono la modifica atomica di più documenti (ordine, prenotazione, template).

Il progetto utilizza **Mongoose** nella versione `6.8.0` come ODM (Object Document Mapper) per TypeScript/JavaScript. Mongoose fornisce un livello di astrazione sopra il driver nativo di MongoDB, offrendo validazione degli schemi, middleware (pre/post hooks), popolamento delle reference e un'API di query fluente.

La configurazione del database in ambiente di sviluppo prevede un **Replica Set** con un singolo nodo, necessario per abilitare il supporto alle transazioni multi-documento. In produzione, il database è ospitato su un'istanza MongoDB gestita con replica set completo.

3.4 Servizi Cloud AWS

L'infrastruttura di Public Pressure è interamente ospitata su **Amazon Web Services (AWS)**, utilizzando diversi servizi gestiti:

- **Amazon ECS (Elastic Container Service)**: servizio di orchestrazione dei container utilizzato per eseguire i servizi del backend. Ogni servizio è distribuito come task ECS su AWS Fargate, che elimina la necessità di gestire i server sottostanti.
- **Amazon SQS (Simple Queue Service)**: servizio di code di messaggi completamente gestito, utilizzato per la comunicazione asincrona tra i diversi servizi. Il progetto utilizza oltre 40 code SQS, organizzate per funzionalità (mail, notifiche, ordini completati, deploy dei contratti) e per blockchain (una coda per ogni rete supportata per i pagamenti e le aste).
- **Amazon S3 (Simple Storage Service)**: servizio di storage di oggetti utilizzato per memorizzare gli asset multimediali (immagini di copertina, file audio, video) e i metadati IPFS.
- **Amazon CloudFront**: servizio di Content Delivery Network (CDN) utilizzato per distribuire gli asset statici con bassa latenza a livello globale.
- **AWS Secrets Manager**: servizio per la gestione sicura dei segreti (chiavi API, credenziali database, chiavi private) con rotazione automatica e controllo degli accessi.
- **Amazon SES e SNS**: servizi per l'invio di email (Simple Email Service) e SMS (Simple Notification Service), utilizzati per le comunicazioni con gli utenti (verifica email, notifiche, OTP).
- **AWS KMS (Key Management Service)**: servizio per la gestione delle chiavi crittografiche utilizzato per la cifratura dei dati sensibili.

3.5 IPFS e Archiviazione dei Metadati

Cos'è IPFS

IPFS (InterPlanetary File System) è un protocollo di rete peer-to-peer per l'archiviazione e la condivisione di contenuti distribuiti. A differenza del Web tradizionale, dove un file è identificato dalla sua *posizione* (un URL che punta a un server specifico), IPFS identifica i file in base al loro *contenuto*: ogni risorsa riceve un identificatore univoco chiamato **CID** (Content Identifier), calcolato come funzione hash del contenuto stesso. Questo meccanismo, noto come *content addressing*, ha due conseguenze fondamentali:

- **Immutabilità**: modificare anche un solo bit del file produce un CID completamente diverso. Chiunque recuperi un file dato il suo CID ha la garanzia crittografica che il contenuto corrisponda a ciò che era stato originariamente archiviato.
- **Assenza di un punto centrale**: il file può essere servito da qualsiasi nodo della rete IPFS che ne conserva una copia, non da un unico server controllato da un soggetto specifico.

Perché IPFS è rilevante per gli NFT

Uno smart contract ERC-721 assegna ad ogni token un URI di metadati tramite la funzione `tokenURI()`. Questi metadati sono un documento JSON che descrive il token: nome, descrizione, URL dell'immagine di copertina, URL del file audio, attributi, ecc. Il problema nasce quando questo URI punta a un server HTTP tradizionale: il proprietario del server potrebbe modificare o rimuovere il file in qualsiasi momento *dopo* la vendita del token, tradendo le aspettative dell'acquirente. Questo tipo di comportamento, noto nel settore come *metadata rug pull*, rappresenta una delle criticità più note degli NFT di prima generazione.

Utilizzare un CID IPFS come `tokenURI` risolve il problema alla radice: il CID è scritto nello smart contract al momento del mint e non può essere alterato; chiunque voglia verificare che i metadati non siano stati manomessi può farlo calcolando l'hash del file recuperato e confrontandolo con il CID registrato on-chain.

Utilizzo in Public Pressure

In Public Pressure, IPFS è utilizzato per archiviare i metadati di ogni NFT musicale e i file media associati (immagine di copertina, audio, documenti di licenza). Il processo è gestito da un microservizio dedicato, `ipfs-assets-uploader`, che viene eseguito

al momento della pubblicazione di un nuovo template da parte di un artista. Il flusso operativo è il seguente:

1. I file originali (immagine, audio in formato WAV o MP3, licenze) vengono **scaricati** dalla sorgente fornita dall'artista.

2. Ogni file viene **caricato su IPFS** tramite le API di **Infura** (un servizio gestito di pinning IPFS), che restituisce il relativo CID.

3. Una copia di ciascun file viene contestualmente **caricata su AWS S3** al percorso `ipfs/{cid}`, in modo che la piattaforma possa servirli rapidamente tramite CloudFront senza dipendere dalla disponibilità della rete IPFS.

4. Viene costruito e caricato su IPFS un **documento JSON di metadati** contenente i CID dell'immagine e dell'audio, la descrizione, il nome, le informazioni sull'artista e il link esterno alla pagina del token.

5. I CID ottenuti vengono **salvati in MongoDB** sul documento del template.

6. Al momento del deploy dello smart contract, il CID del documento JSON viene passato come `baseURI_` al costruttore del contratto. La funzione `tokenURI()` del contratto restituirà questo URI per ogni token coniato.

Il pattern doppio storage: IPFS + S3

Un aspetto architetturale rilevante è la scelta di mantenere le risorse sia su IPFS che su S3. I due sistemi svolgono ruoli complementari e distinti:

- **IPFS** garantisce l'*integrità* e la *verificabilità* del contenuto. Il CID registrato nello smart contract è una prova crittografica permanente di cosa debba contenere quel token.

- **S3 + CloudFront** garantisce la *disponibilità* e le *prestazioni* di accesso. La rete IPFS pubblica non offre garanzie di latenza o uptime; un file può essere recuperato lentamente se pochi nodi ne conservano una copia. Tramite CloudFront, il browser dell'utente riceve le risorse da un edge node geograficamente vicino in pochi millisecondi.

Il percorso S3 che replica la struttura IPFS (`ipfs/{cid}`) non è casuale: consente di mantenere una corrispondenza diretta tra l'identificatore IPFS e la risorsa su S3, semplificando la verifica e la gestione. L'URL pubblico CloudFront e il CID IPFS fanno riferimento esattamente allo stesso contenuto.

Infura come servizio di pinning

IPFS non garantisce di per sé la persistenza dei dati: un file rimane disponibile sulla rete solo finché almeno un nodo lo mantiene in memoria (*pinning*). Il progetto utilizza **Infura** come servizio di pinning gestito: caricare un file tramite le API di Infura implica che Infura si faccia carico di mantenerlo disponibile sulla rete IPFS, senza necessità di gestire un nodo proprio.

3.6 Containerizzazione e Orchestrazione

Docker

Docker [18] è la tecnologia di containerizzazione utilizzata per impacchettare e distribuire tutti i servizi del progetto. Ogni servizio ha il proprio `Dockerfile` che definisce un processo di build multi-stage:

1. **Stage deps**: installazione delle dipendenze di base.
2. **Stage builder**: compilazione del codice TypeScript e dei contratti Solidity.
3. **Stage runtime**: immagine finale minimale con solo le dipendenze di produzione e il codice compilato.

Questo approccio multi-stage riduce significativamente la dimensione delle immagini finali, eliminando le dipendenze di sviluppo e i file sorgente dalla produzione. Le immagini base utilizzano `node:18.5-alpine3.15`, una distribuzione Linux minimale che riduce ulteriormente la dimensione dell'immagine finale.

Per lo sviluppo locale, il progetto utilizza **Docker Compose** per orchestrare i servizi necessari: MongoDB (configurato come Replica Set), LocalStack (per l'emulazione locale di AWS SQS) e i servizi di blockchain-pulling per le reti di test.

LocalStack

LocalStack è un emulatore dei servizi AWS utilizzato in ambiente di sviluppo per simulare le code SQS senza incorrere in costi di cloud. La configurazione del progetto prevede l'inizializzazione automatica di 46 code (7 code generiche con relative dead-letter queue e 3 code per ciascuna delle 4 reti blockchain supportate in sviluppo).

3.7 Strumenti di Sviluppo e Collaborazione

Git e GitHub

Il codice sorgente del progetto è gestito tramite **Git**, con il repository ospitato su **GitHub**. Il flusso di lavoro Git adottato prevede due branch principali ed altri secondari:

- `function_branch`: branch creati per lavorare ad una funzionalità specifica. Quando l'implementazione è completa e pronta per essere testata, i contenuti del branch vengono aggiunti al branch dev in modo da poter testare l'implementazione.
- `dev`: branch di integrazione per l'ambiente di sviluppo. Ogni push su questo branch avvia automaticamente la pipeline CI/CD che esegue linting, test e deploy sull'ambiente di sviluppo.
- `prod`: branch di produzione. Le modifiche vengono promosse da `dev` a `prod` dopo la validazione nell'ambiente di sviluppo.

Al completamento di una nuova funzionalità, questa non viene immediatamente integrata sul branch dev dallo sviluppatore che ha eseguito il lavoro, ma viene avviato un processo di review dell'implementazione chiamato pull request (PR). Una PR viene creata con la richiesta di integrare per esempio i contenuti di **function_branch** sul branch **dev**. La PR viene revisionata da un secondo sviluppatore (solitamente con più seniority del creatore della richiesta) che può (eventualmente) dare indicazioni su cosa sia necessario cambiare prima che l'implementazione possa essere integrata, oppure approvare la richiesta e quindi chiuderla.

Per la gestione dei task, il team utilizza **GitHub Projects**, uno strumento che permette di visualizzare le issue in formato task board, assegnare persone e scadenze e categorizzare i task in base allo stato di avanzamento.

Yarn Workspaces e Struttura Monorepo

Il progetto adotta un'architettura a **monorepo** (monolithic repository) gestita tramite **Yarn 3.2.0** con il plugin workspace-tools. Una monorepo è una repository che contiene il codice di più progetti o pacchetti correlati, consentendo la condivisione di codice e la gestione unificata delle dipendenze. Una multi-repo invece prevede una repository separata per ogni componente del progetto.

I pacchetti principali della monorepo sono elencati nella tabella 3.1.

Tabella 3.1: Pacchetti principali della repository

Pacchetto	**Descrizione**
`@pp/be`	Backend principale (API REST, gestione ordini, pagamenti)
`@pp/cms`	Content Management System per operatori e admin
`@pp/blockchain`	Interazione con le blockchain (deploy, minting, aste)
`@pp/blockchain-pulling`	Servizio di sincronizzazione blockchain $\rightarrow$ database
`@pp/mongo`	Modelli MongoDB condivisi (Mongoose schemas)
`@pp/contracts`	Codice sorgente dei contratti Solidity
`@pp/libs`	Librerie condivise (autenticazione, utility, middleware)
`@pp/openapi`	Generazione delle specifiche OpenAPI
`@pp/transports`	Layer di comunicazione (email, notifiche)
`@pp/auto-import`	Plugin di auto-caricamento delle route Fastify

La struttura a monorepo offre vantaggi significativi: le modifiche che impattano più servizi (ad esempio, una modifica a un modello MongoDB) possono essere effettuate in un singolo commit, garantendo la coerenza tra tutti i pacchetti che utilizzano quel modello.

Comunicazione del Team

La comunicazione all'interno del team di sviluppo è avvenuta attraverso **Slack** per le comunicazioni quotidiane e **Discord** per le riunioni e le discussioni tecniche. Le riunioni periodiche (sprint planning, daily standup, retrospettive) sono state condotte seguendo la metodologia Scrum, con sprint della durata di quattro settimane.

ESLint e Qualità del Codice

La qualità del codice è garantita da una configurazione rigorosa di **ESLint**, il linter standard per JavaScript/TypeScript. La configurazione del progetto include oltre 50 regole attive, tra cui:

- Indentazione a 4 spazi (errore).
- Apici singoli obbligatori (warning).
- Punto e virgola obbligatorio (errore).

- Profondità massima di annidamento: 4 livelli.
- Numero massimo di callback annidati: 3.
- Controllo delle promise non gestite (`@typescript-eslint/no-floating-promises`).

Husky è utilizzato per configurare un hook `pre-commit` che esegue automaticamente ESLint con auto-fix sui file modificati prima di ogni commit, impedendo che codice non conforme agli standard venga inserito nella repository.

Capitolo 4

Architettura del Sistema

Questo capitolo presenta il design architetturale complessivo di Public Pressure, descrivendo la struttura orientata a servizi indipendenti, l'organizzazione della repository, il modello dei dati e i flussi di comunicazione tra i componenti.

Prima di analizzare l'architettura, la Tabella 4.1 fornisce una panoramica delle dimensioni del progetto.

Tabella 4.1: Dimensioni del progetto Public Pressure

Metrica	**Valore**
Linee di codice totali (TypeScript, JavaScript, Solidity)	∼82 700
Numero di file sorgente	787
Pacchetti nella monorepo	17
Modelli MongoDB (schemi Mongoose)	51
Contratti Solidity	35
Moduli API (route)	20
Endpoint REST	165
Code SQS	46
Workflow GitHub Actions	15
File di test	51
Test case totali	974
Blockchain supportate (produzione + test)	9

4.1 Architettura orientata ai servizi (SOA)

Public Pressure adotta un'architettura orientata ai servizi (SOA) in cui le diverse funzionalità del marketplace sono distribuite su servizi indipendenti, ciascuno con il proprio

ciclo di vita, il proprio processo di deploy e le proprie risorse.

I servizi principali del sistema sono:

1. **Backend (BE)**: il servizio principale che espone le API REST per il frontend, gestisce l'autenticazione degli utenti, il catalogo NFT, gli ordini, i pagamenti e la comunicazione con i servizi esterni. Ascolta sulla porta 8080.

2. **Content Management System (CMS)**: un servizio dedicato alle operazioni amministrative, utilizzato dagli operatori e dagli artisti per gestire collezioni, template NFT, whitelist, categorie e report. Espone API separate sulla porta 4000.

3. **Blockchain Service**: uno o più servizi (uno per ogni blockchain supportata) che si occupano dell'interazione diretta con le blockchain: deploy dei contratti, minting degli NFT, gestione delle aste e delle vendite a prezzo fisso.

4. **Blockchain-Pulling Service**: uno o più servizi (uno per ogni blockchain supportata) che monitorano continuamente le blockchain per cercare eventi rilevanti (trasferimenti, aste, vendite) e aggiornare il database dell'applicazione.

5. **Servizi di supporto**: servizi schedulati per operazioni periodiche come la generazione di report (`report-job`), l'invio di reminder agli artisti (`artist-reminder`), il caricamento degli asset su IPFS (`ipfs-assets-uploader`) e l'aggiornamento dei dati statistici (`charts-push`).

La Figura 4.1 illustra l'architettura complessiva del sistema e i flussi di comunicazione tra i servizi.

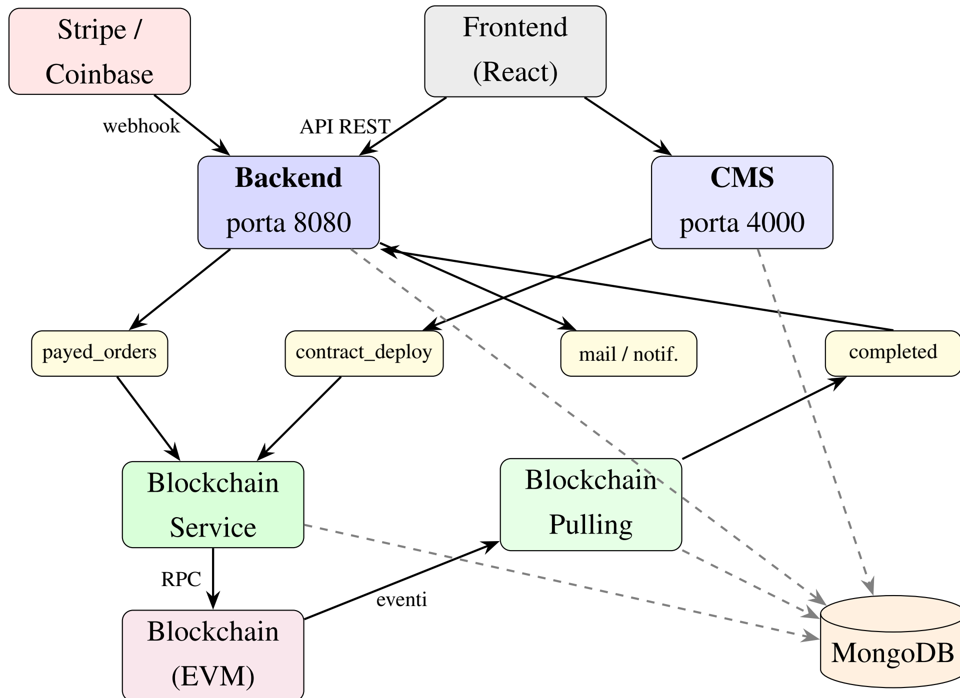

Figura 4.1: Architettura a servizi di Public Pressure con i flussi di comunicazione via SQS e MongoDB condiviso.

Vantaggi dell'Architettura a Servizi

La scelta di un'architettura a servizi singoli ed indipendenti, rispetto a un'applicazione monolitica, è motivata da:

- **Scalabilità indipendente**: ogni servizio può essere scalato in modo indipendente. Ad esempio, durante un drop importante, il backend può essere scalato orizzontalmente senza dover scalare anche il CMS o i servizi blockchain.
- **Isolamento dei guasti**: un problema in un servizio non compromette l'intero sistema. Se il servizio di blockchain-pulling per Polygon si blocca, le operazioni su Moonbeam continuano a funzionare normalmente.
- **Deploy indipendente**: ogni servizio può essere aggiornato in modo indipendente, riducendo il rischio associato ai rilasci e accelerando il ciclo di sviluppo.
- **Separazione delle responsabilità**: ogni servizio ha un dominio ben definito, facilitando la manutenzione e la comprensione del codice.

Comunicazione tra Servizi

La comunicazione tra i diversi servizi avviene attraverso due canali principali:

- **Code di messaggi (AWS SQS)**: per la comunicazione asincrona. Quando il backend riceve un pagamento confermato, invia un messaggio sulla coda corrispondente alla blockchain dell'ordine. Il servizio blockchain consuma il messaggio e procede al minting dell'NFT. Questo pattern disaccoppia temporalmente i servizi e garantisce la consegna dei messaggi anche in caso di indisponibilità temporanea del consumatore.
- **Database condiviso (MongoDB)**: i servizi condividono l'accesso al database tramite i modelli definiti nel pacchetto `@pp/mongo`. Questa scelta, sebbene introduca un accoppiamento a livello di dati, è motivata dalla necessità di coerenza dei dati tra i servizi e dalla complessità che un'architettura completamente event-driven avrebbe introdotto.

L'architettura delle code SQS è organizzata in due categorie come descritto nelle tabelle 4.2 e 4.3.

Tabella 4.2: Code SQS generiche

Coda	**Funzione**
`mail`	Invio email (verifica, notifiche, conferme)
`notification`	Notifiche in-app per gli utenti
`completed_orders`	Ordini completati con successo
`failed_orders`	Ordini falliti
`deployed_contract`	Conferma di deploy di un contratto
`auction_error`	Errori nel processo di asta
`kilt_assetdid`	Operazioni di identità decentralizzata KILT

Tabella 4.3: Code SQS per blockchain (replicate per ogni rete)

Coda	**Funzione**
`payed_orders_{chain}`	Ordini pagati da processare (minting)
`contract_deployment_{chain}`	Richieste di deploy di nuovi contratti
`auction_{chain}`	Operazioni relative alle aste

Ogni coda ha una corrispondente **dead-letter queue** (DLQ) che raccoglie i messaggi che non sono stati elaborati con successo dopo un numero massimo di tentativi, permettendo l'analisi e il recupero manuale degli errori.

4.2 Modello dei Dati

Il database MongoDB di Public Pressure è organizzato in diverse collezioni, ciascuna gestita da un modello Mongoose definito nel pacchetto `@pp/mongo`. Di seguito vengono descritti i modelli principali.

Modello User

Il modello `User` rappresenta un utente della piattaforma e contiene:

- **Credenziali**: email (unica, obbligatoria), username (unico), password (hash argon2).
- **Dati personali**: nome, cognome, data di nascita, genere, paese, dati aziendali (per utenti business).
- **Verifica**: stato di conferma email e mobile, codici di verifica (2FA).
- **Wallet**: array di indirizzi blockchain associati, con indicazione dell'indirizzo di default e degli indirizzi custodian.
- **Ruoli**: associazioni con artisti e label, ciascuna con un ruolo (`owner`, `editor`, `admin`).
- **Preferenze**: valuta preferita, impostazioni di notifica, newsletter.
- **NFT favoriti**: riferimenti ai template NFT salvati come preferiti.

Modello Template

Il modello `Template` è il più complesso del sistema e rappresenta un "tipo" di NFT (ad esempio, un brano musicale in edizione limitata a 100 copie). I campi principali includono:

- **Stato**: un campo enum che traccia il ciclo di vita del template: `draft` → `awaiting-internal-approval` → `awaiting-parts-approval` → `to-be-deployed` → `deployment` → `deployed` → `active` → `paused` → `expired`.
- **Deployment**: array di oggetti che tracciano il deploy del contratto su ciascuna blockchain, con indirizzo del contratto, stato del deploy e metadati IPFS.
- **Contenuto**: titolo, descrizione, tipo di media (`music`, `video`, `collectible`), link al file multimediale, immagine di copertina.

- **Vendita**: tipo di vendita (`auction`, `fixed`, `airDrop`, `notForSale`), prezzo in USD, prezzi in crypto, date di inizio e fine (valido per tutti i tipi di vendita), numero di edizioni.
- **Royalty**: array di destinatari con le relative percentuali, stati di approvazione e indirizzi wallet.
- **Whitelist**: configurazione dell'accesso anticipato per acquisto o claim gratuito.
- **Generi e categorizzazione**: generi musicali (25 possibili), tipologia artistica, contenuti riscattabili (merchandise fisico).

Il ciclo di vita di un template è mostrato nella figura 4.2, ovvero viene descritto il cambiamento dello stato del template dalla sua creazione nello stato **draft** fino allo stato **expired**. Lo stato **paused** è particolare in quanto non è parte del flusso principale, ma è uno stato particolare che può essere assunto temporaneamente da un template (e dallo smart contract di riferimento). Questo ha a che fare con un sistema di sicurezza degli smart contract che verrà discusso più in dettaglio nel capitolo 6.

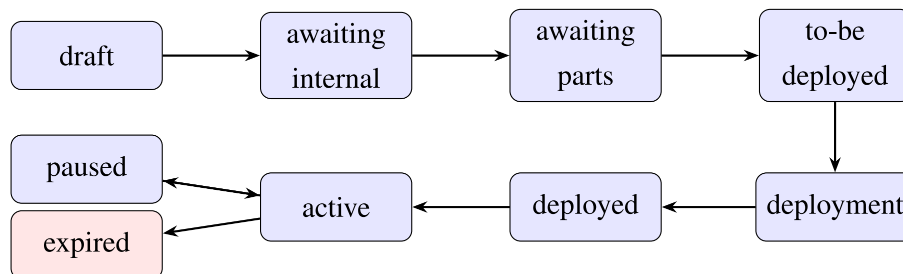

Figura 4.2: Macchina a stati del modello Template.

Modello Order

Il modello `Order` traccia ogni transazione di acquisto e implementa una macchina a stati, come mostrato nella figura 4.3:

- `created`: l'ordine è stato creato e l'edizione è stata prenotata.
- `payed`: il pagamento è stato confermato (da Stripe, Coinbase o on-chain).
- `completed`: è stato fatto il minting dell'NFT ed è stato trasferito al compratore.
- `failed`: il pagamento o il minting sono falliti.
- `cancelled`: l'ordine è stato annullato dall'utente o dal sistema.

Ogni ordine registra il gateway di pagamento utilizzato (`stripe`, `cc`, `erc20`, `free`), l'indirizzo blockchain del compratore, la blockchain di destinazione, l'ID del template e, al completamento, l'hash della transazione di minting.

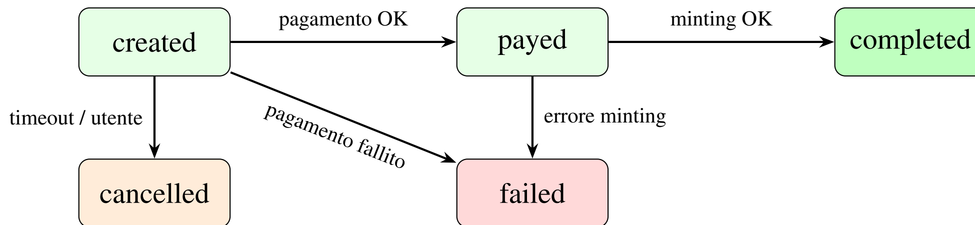

Figura 4.3: Macchina a stati del modello Order.

Modello Artist e Label

I modelli `Artist` e `Label` rappresentano rispettivamente artisti individuali ed etichette discografiche. Entrambi contengono informazioni di profilo (username, immagini, descrizione, social media), la lista dei collaboratori con i relativi ruoli (owner, editor oppure admin), lo stato di verifica dell'account Stripe per la ricezione dei pagamenti e i dati bancari per i pagamenti fuori piattaforma. Una **Label** può avere anche una lista di **Artists** associti alla label.

Altri Modelli Rilevanti

- **Edition**: rappresenta una singola edizione (copia) di un NFT, con tokenId, indirizzo del contratto, proprietà e prezzo di vendita.
- **Reservation**: traccia le prenotazioni temporanee delle edizioni durante il processo di acquisto, con contatori atomici per gestire la concorrenza.
- **Auction**: metadati delle aste (auctionId, offerta corrente, offerenti, stato).
- **BidHistory**: storico delle offerte per ogni asta.
- **NFTHistory**: storico completo dei trasferimenti e delle transazioni per ogni NFT. Questo ci permette di mostrare all'utente lo storico completo della vita di un NFT senza la necessità di interrogare la blockchain per ottenere queste informazioni.
- **FiatSale**: record delle vendite in valuta FIAT con dettagli su importi lordi, commissioni e royalty distribuite.
- **Accountability**: registro contabile per il tracciamento delle royalty dovute e pagate.

- **Drop**: collezione di template raggruppati per un evento di lancio specifico.
- **EarlyAccessList**: liste di utenti con accesso anticipato a specifici drop o template.

La Figura 4.4 mostra le relazioni principali tra i modelli del database. Dalla figura si può notare la relazione Artist-User N:M, questa relazione riguarda soltanto gli account utente collaboratori di un certo artista. Ogni entità artista ha almeno un collaboratore mentre ogni utente può essere collaboratore di nessuno, uno o più artisti.

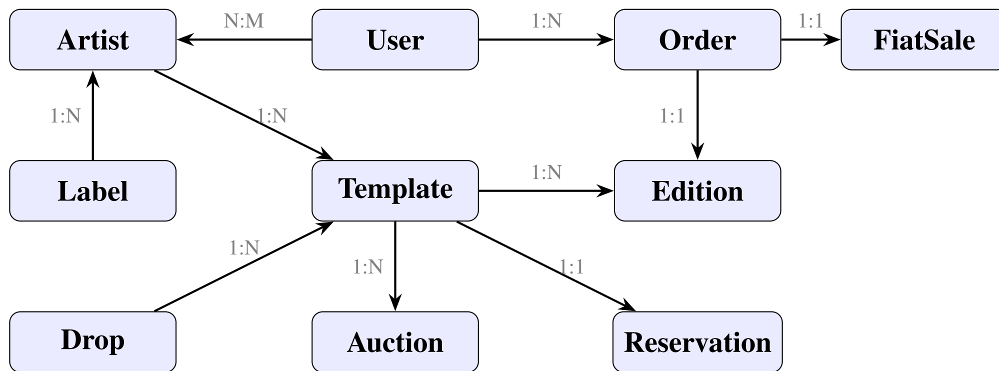

Figura 4.4: Diagramma ER semplificato delle relazioni tra le entità principali del database.

4.3 Flusso Operativo del Sistema

Il flusso operativo principale del marketplace può essere descritto attraverso i seguenti scenari.

Creazione e Pubblicazione di un NFT

1. L'artista o l'etichetta crea un nuovo template tramite il CMS, specificando i metadati, il file multimediale, il prezzo, il tipo di vendita e le royalty.
2. Il template attraversa un processo di approvazione interna e, se coinvolge più artisti, un'approvazione delle parti.
3. Una volta approvato, il template viene marcato come "to-be-deployed" e il CMS invia un messaggio sulla coda `contract_deployment` per ogni blockchain su cui deve essere eseguito il deploy.
4. Il servizio blockchain riceve il messaggio, compila ed esegue il deploy del contratto `ERC721Multiple` sulla blockchain corrispondente, configurando le royalty ERC-2981 e registrando il contratto nel Diamond (GBM).

5. Al completamento del deploy, il servizio blockchain invia un messaggio sulla coda `deployed_contract` con l'indirizzo del contratto. Il backend aggiorna il template con i dati del deployment.

6. Gli asset multimediali vengono caricati su IPFS tramite il servizio `ipfs-assets-uploader`, e i CID IPFS vengono salvati nei metadati del template.

Acquisto di un NFT (Vendita a Prezzo Fisso)

1. L'utente seleziona un NFT e sceglie il metodo di pagamento (carta, Coinbase, USDC).

2. Il frontend invia una richiesta POST al backend con i dettagli dell'ordine.

3. Il backend, all'interno di una transazione MongoDB:

   - Crea un documento `Order` con stato `created`.
   - Incrementa atomicamente il contatore `reserved` nel documento `Reservation` corrispondente, verificando che non superi il numero di edizioni disponibili.
   - Se il gateway è Stripe, crea una sessione di checkout e restituisce l'URL.
   - Se il gateway è Coinbase, crea un charge e restituisce l'URL della pagina di pagamento.
   - Se il gateway è ERC-20, genera una firma crittografica per l'autorizzazione del pagamento on-chain.

4. Al completamento del pagamento (confermato tramite webhook), il backend:

   - Aggiorna l'ordine a `payed`.
   - Calcola e registra le commissioni della piattaforma e le royalty degli artisti.
   - Invia un messaggio sulla coda `payed_orders_{chain}` per il minting.

5. Il servizio blockchain riceve il messaggio e procede al minting dell'NFT sul contratto corrispondente.

6. Il servizio blockchain-pulling rileva l'evento `Transfer` sulla blockchain e:

   - Crea un record in `NFTHistory`.
   - Aggiorna lo stato della transazione.
   - Crea la relazione fra l'edizione e l'utente.
   - Invia un messaggio di completamento sulla coda `completed_orders`.

7. Il backend riceve il messaggio di completamento e finalizza l'ordine: aggiorna lo stato a `completed`, notifica l'utente che l'acquisto è andato a buon fine ed infine verifica se il template è sold-out.

La Figura 4.5 riassume il flusso di acquisto.

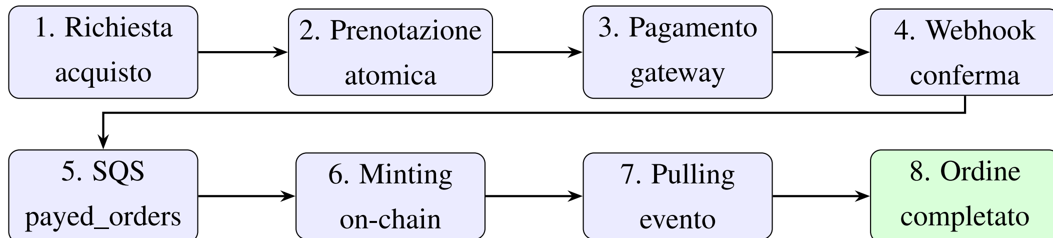

Figura 4.5: Flusso sequenziale dell'acquisto di un NFT a prezzo fisso, dalla richiesta iniziale al completamento.

Asta GBM

1. L'artista o la piattaforma registra un'asta sul Diamond contract tramite la funzione `registerAnAuctionToken`, specificando i parametri dell'asta (durata, incremento minimo, incentivi).

2. Gli utenti piazzano offerte tramite il frontend, che interagisce direttamente con il contratto GBM sulla blockchain. Ogni offerta deve rispettare l'incremento minimo e superare l'offerta corrente.

3. Quando un'offerta viene superata, il contratto rimborsa automaticamente l'offerente precedente con l'aggiunta dell'incentivo calcolato.

4. Se un'offerta viene piazzata negli ultimi minuti dell'asta (Hammer Time), la durata viene estesa automaticamente.

5. Alla scadenza dell'asta, dopo il periodo di grazia, chiunque può chiamare la funzione `claim` che trasferisce l'NFT al vincitore e distribuisce i proventi (al venditore, alla piattaforma come commissione, ed agli artisti come royalty).

6. Il servizio blockchain-pulling rileva gli eventi dell'asta e aggiorna il database con lo storico delle offerte, il risultato dell'asta e i trasferimenti degli NFT.

Capitolo 5

Implementazione del Backend

Questo capitolo descrive in dettaglio l'implementazione del backend di Public Pressure, con focus sulla struttura delle API, il sistema di autenticazione, la gestione degli ordini e le integrazioni con i servizi esterni.

5.1 Struttura dell'Applicazione Fastify

L'applicazione Fastify è configurata nel file `app.ts`, che rappresenta il punto di ingresso del backend. L'inizializzazione segue un ordine preciso:

1. **Creazione dell'istanza Fastify** con configurazione del logger (Pino) e del timeout keep-alive (75 secondi in produzione).

2. **Registrazione del parser raw body** per la cattura del body non elaborato delle richieste, necessario per la verifica delle firme dei webhook Stripe.

3. **Configurazione della sicurezza**: registrazione di `@fastify/helmet` per gli header di sicurezza HTTP (Content Security Policy, Cross-Origin Resource Policy, ecc.) e di `@fastify/cors` per la gestione delle richieste cross-origin.

4. **Configurazione della validazione**: compilazione e registrazione degli schemi JSON globali tramite AJV (Another JSON Validator), utilizzati per la validazione automatica di input e output delle API.

5. **Configurazione di Swagger**: solo in ambiente di sviluppo, registrazione di `@fastify/swagger` per la generazione automatica della documentazione OpenAPI 3.1.0, accessibile all'endpoint `/main/docs`.

6. **Registrazione delle route**: caricamento automatico di tutti i moduli API dalla directory `src/v1` tramite il plugin `@pp/auto-import`.

7. **Configurazione degli hook globali**: registrazione dell'hook `onResponse` per il logging strutturato di tutte le richieste (escludendo OPTIONS, HEAD, documentazione e healthcheck).

Sistema a Plugin e Auto-Import

La struttura delle route sfrutta il sistema a plugin di Fastify per creare un albero gerarchico di incapsulamento. Il plugin `@pp/auto-import` scansiona automaticamente la directory `src/v1` e registra ogni file `routes.ts` trovato come plugin Fastify con il prefisso corrispondente al nome della directory.

Ogni modulo API segue un pattern standardizzato che separa le route pubbliche (senza autenticazione), le route protette (con JWT) e le route con requisiti specifici di ruolo. Nel listato 5.1 si possono vedere alcuni esempi di route protette da uno o più middleware. Per esempio una chiamata alla route `nft` deve passare i controlli di due funzioni middleware: **middlewareJWT** (controlla che la chiamata abbia un JWT e che questi sia valido) ed anche **artistRole** (controlla che l'utente che ha effettuato la chiamata abbia il ruolo artist, il ruolo se presente viene salvato nel JWT alla sua generazione in fase di login).

Questo pattern garantisce che i controlli vengano applicati solo alle route appropriate: le route pubbliche hanno la protezione reCAPTCHA ma non richiedono autenticazione, mentre le route protette richiedono un token JWT valido.

La Figura 5.1 illustra il ciclo di vita di una richiesta HTTP.

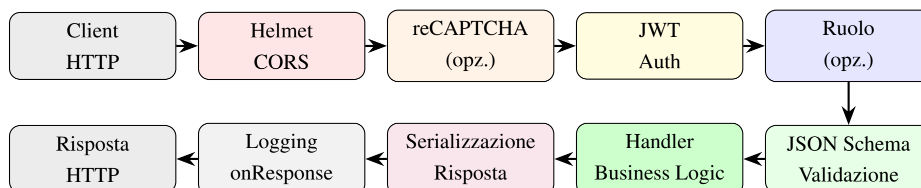

Figura 5.1: Ciclo di vita di una richiesta nel backend Fastify, attraverso i layer di sicurezza, autenticazione, validazione e risposta.

```
// Route pubbliche con reCAPTCHA
const openAPI = async (fastify) => {
    await fastify.register(middlewareRecaptcha);
    fastify.post('/login', loginHandler);
    fastify.post('/register', registerHandler);
};

// Route protette con JWT
const authAPI = async (fastify) => {
    await fastify.register(middlewareJWT);
    fastify.get('/profile', getProfileHandler);
    fastify.patch('/profile', updateProfileHandler);
};

// Route con ruolo artista
const artistAPI = async (fastify) => {
    await fastify.register(middlewareJWT);
    await fastify.register(artistRole);
    fastify.post('/nft', createNFTHandler);
};

// Registrazione con prefisso
const plugin = async (fastify) => {
    await fastify.register(async (f) => {
        await f.register(openAPI);
        await f.register(authAPI);
        await f.register(artistAPI);
    }, { prefix: '/user' });
};
```

Listato 5.1: Pattern di organizzazione delle route in un modulo API

5.2 Sistema di Autenticazione e Autorizzazione

Il sistema di autenticazione di Public Pressure è basato su **JSON Web Token (JWT)** e implementa un modello di autorizzazione a ruoli gerarchico.

Autenticazione JWT

L'autenticazione avviene tramite un token JWT firmato con un segreto condiviso (algoritmo configurabile tramite variabile d'ambiente). Il token contiene l'identificatore dell'utente (`_id`) e ha una scadenza configurabile.

Il processo di autenticazione segue questi passi:

1. L'utente effettua il login fornendo email e password.
2. Il backend verifica le credenziali: la password viene confrontata con l'hash argon2 memorizzato nel database.
3. Se le credenziali sono valide, viene generato un token JWT che viene restituito al client.
4. Il client include il token nell'header `Authorization: Bearer {token}` di ogni richiesta successiva.
5. Il middleware `middlewareJWT` verifica la validità del token e decora la richiesta con i dati dell'utente (`request.jwt`).

Middleware di Autorizzazione

Il sistema implementa dieci middleware di autorizzazione distinti, ciascuno per un contesto specifico:

- **middlewareJWT**: verifica base del token JWT. Tutte le route protette lo utilizzano.
- **artistRole / labelRole**: verifica che l'utente abbia un ruolo associato all'artista o all'etichetta specificata nell'header `x-artist-id` o `x-label-id`. Controlla che l'utente sia elencato tra i collaboratori dell'entità.
- **ownerRole**: verifica che l'utente sia il proprietario (ruolo `owner`) dell'artista o dell'etichetta. Utilizzato per operazioni critiche come la modifica dei pagamenti.

- **templateRole / collectionRole / dropRole**: riservati ai super-utenti della piattaforma. Verificano che l'utente sia nella lista dei super-utenti configurata nelle variabili d'ambiente.
- **middleware2FA**: verifica l'autenticazione a due fattori tramite codice OTP inviato via SMS. L'utente deve includere il codice nell'header `x-code`.
- **earlyAccessListMiddleware**: verifica se l'utente ha accesso anticipato a un determinato drop o template, decorando la richiesta con la funzione `hasEarlyAccessTo`.

Il listato 5.2 mostra l'implementazione del middleware JWT e del middleware per il ruolo artista/etichetta, estratti dal file `authentication.ts`:

```
const middlewareJWT = fp(async (fastify, opts) => {
    fastify.decorateRequest('jwt', null);
    fastify.addHook('preHandler', async (request, reply) => {
        request.jwt = await authenticationClient
            .verifyJWT(request);
    });
});

const labelOrArtistRole = fp(async (fastify, opts) => {
    fastify.decorateRequest('caller', null);
    fastify.addHook('preHandler', async (request, reply) => {
        const artistId = request.headers['x-artist-id'];
        const labelId = request.headers['x-label-id'];
        if (!artistId && !labelId)
            return reply.badRequest('NO_ARTIST_ID_NOR_LABEL_ID');
        if (artistId) {
            const user = await User.findById(
                request.jwt._id, { artists: true }
            ).lean();
            const found = user.artists
                .find(e => e._id === artistId);
            if (!found)
                return reply.forbidden(
                    'ARTIST_NOT_MANAGED_BY_USER');
            request.caller = {
                ...found, entityKind: 'artist'
            };
        }
    });
});
```

Listato 5.2: Middleware JWT e verifica del ruolo artista/etichetta

La Figura 5.2 mostra la gerarchia dei middleware di autorizzazione e come si compongono. A sinistra: `recaptchaAPI` per route pubbliche con captcha. A destra: la catena per route protette, sempre nell'ordine JWT → ruolo entità → restrizione.

`earlyAccessListMiddleware` (tratteggiato) è un decoratore di scope registrato parallelamente a JWT, non in sequenza. Non tutte le route sono precedute da tutti i controlli, alcune prevedono soltanto `middlewareJWT` per esempio.

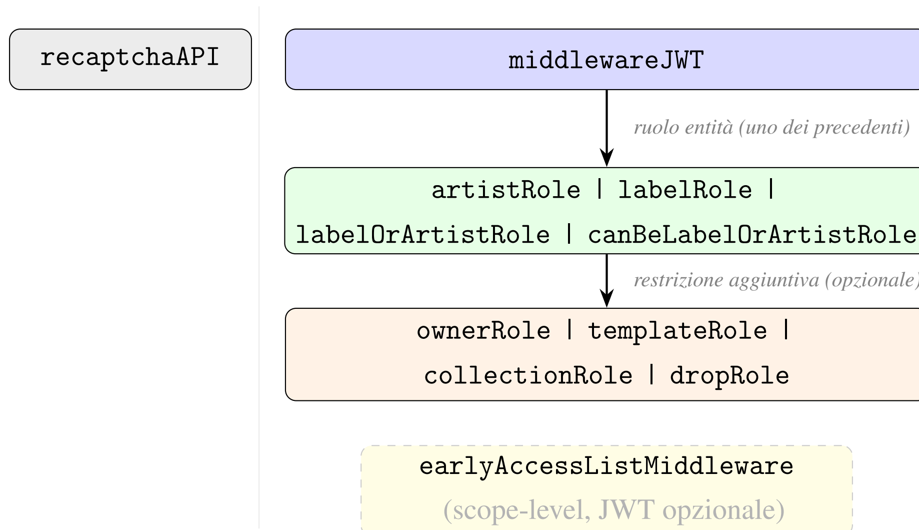

Figura 5.2: Composizione dei middleware di autenticazione.

5.3 Validazione dei Dati e JSON Schema

Public Pressure adotta un approccio di validazione rigoroso basato su **JSON Schema**. Nel listato 5.3 viene mostrato un esempio di schema che definisce la struttura di `querystring, body e response` (querystring referenzia una definizione importata, spesso usate per essere referenziate in più schemi). Ogni endpoint API definisce schemi per:

- **Querystring**: parametri nell'URL.
- **Body**: corpo della richiesta per le operazioni POST/PUT/PATCH.
- **Params**: parametri nell'URL (ad esempio, `/template/:id`).
- **Response**: schema della risposta per ogni codice HTTP, utilizzato sia per la validazione che per la serializzazione ottimizzata.

La validazione tramite JSON Schema offre diversi vantaggi:

- Rifiuto automatico delle richieste malformate con messaggi di errore dettagliati.
- Prevenzione dell'iniezione di campi non previsti (`additionalProperties: false`).

```
export default route({
    tags: ['payments'],
    querystring: { $ref: 'blockchain' },
    body: {
        type: 'object',
        additionalProperties: false,
        required: ['contractAddress', 'quantity', 'gateway'],
        properties: {
            contractAddress: { type: 'string' },
            quantity: { type: 'integer', minimum: 1, maximum: 12 },
            gateway: {
                type: 'string',
                enum: ['stripe', 'cc', 'free', 'erc20']
            },
            address: { type: 'string' },
            currency: { type: 'string' }
        }
    },
    response: {
        201: {
            type: 'object',
            required: ['order'],
            properties: {
                order: { /* schema ordine */ },
                stripe: { type: 'object', required: ['url'] },
                coinbase: { type: 'object', required: ['hostedUrl'] },
                erc20: { type: 'object', required: ['buySignature'] }
            }
        }
    }
}, handler);
```

Listato 5.3: Esempio di schema per l'endpoint di creazione ordine

- Generazione automatica della documentazione OpenAPI.
- Serializzazione ottimizzata delle risposte (Fastify compila gli schemi in serializzatori veloci).

5.4 Gestione degli Ordini

La gestione degli ordini è il cuore del backend e coordina l'intero flusso di acquisto, dalla creazione alla finalizzazione.

Creazione dell'Ordine e Sistema di Prenotazione

Quando un utente inizia la procedura di acquisto, il backend esegue le seguenti operazioni all'interno di una **transazione MongoDB** (sessione in cui le operazioni eseguite vengono applicate soltanto se tutto va a buon fine, rendendo quindi atomiche le operazioni al suo interno):

1. **Verifica della disponibilità**: controlla che il template sia attivo, non sold-out e che il tipo di vendita sia compatibile con il gateway scelto.
2. **Verifica del gateway**: controlla che il gateway di pagamento sia disponibile per il paese dell'utente e per il tipo di template (ad esempio, i template "curated"(sezione 7.2) non possono essere acquistati tramite Coinbase).
3. **Creazione della prenotazione**: esegue un `findOneAndUpdate` atomico sul documento `Reservation` con la condizione:

```
Reservation.findOneAndUpdate({
    contractAddress: order.contractAddress,
    blockchain: order.blockchain,
    $expr: {
        $gte: [
            { $subtract: ['$edition', '$reserved'] },
            quantity
        ]
    }
}, {
    $inc: { reserved: quantity }
}, { session });
```

Listato 5.4: Prenotazione atomica delle edizioni

Questa operazione è atomica e thread-safe: se due utenti tentano di acquistare l'ultima edizione contemporaneamente, solo uno dei due riuscirà ad incrementare il contatore, mentre l'altro riceverà un errore.

4. **Creazione dell'ordine**: crea il documento `Order` con stato `created`.

5. **Inizializzazione del pagamento**: in base al gateway selezionato, crea la sessione di pagamento e restituisce i dati necessari al frontend.

Cancellazione degli Ordini

Gli ordini possono essere cancellati (dall'utente o automaticamente per timeout) attraverso un processo che:

1. Aggiorna lo stato dell'ordine a `cancelled`.

2. Decrementa atomicamente il contatore `reserved` nel documento `Reservation`, liberando le edizioni per altri acquirenti.

3. Se applicabile, annulla il pagamento sul gateway esterno.

Finalizzazione degli Ordini

Quando il servizio blockchain-pulling rileva l'evento di minting sulla blockchain, invia un messaggio sulla coda `completed_orders`. Il backend processa il messaggio e:

1. Aggiorna l'ordine a `completed` con l'hash della transazione.

2. Aggiorna le whitelist (per template con accesso limitato).

3. Aggiorna le edizioni con il proprietario e il prezzo di vendita.

4. Se l'utente utilizza un indirizzo custodian, aggiorna il flag `hasCustodian`.

5. Verifica se tutte le edizioni sono state vendute e, in caso affermativo, segna il template come `soldOut`.

6. Invia notifiche all'utente (email e notifica in-app).

5.5 Sistema di Code di Messaggi

Il sistema di code è implementato tramite una classe wrapper attorno al client AWS SQS come mostrato nel listato 5.5.

Il pattern **long polling** (con `WaitTimeSeconds = 20`) riduce il numero di richieste vuote a SQS, ottimizzando i costi e riducendo la latenza nel processamento dei messaggi. Il sistema di SQS non prevede il mantenere una connessione attiva verso i consumer

```
class Queue extends EventEmitter {
    constructor(QueueUrl, MaxNumberOfMessages = 1,
                WaitTimeSeconds = 20) {
        this.QueueUrl = QueueUrl;
        this.MaxNumberOfMessages = MaxNumberOfMessages;
        this.WaitTimeSeconds = WaitTimeSeconds;
    }

    async run(callback) {
        while (this.forever) {
            const { Messages } = await sqs.send(
                new ReceiveMessageCommand(this.params)
            );
            if (!Messages) continue;
            const { ReceiptHandle, Body } = Messages[0];
            const { name, params } = JSON.parse(Body);

            await callback(name, params, state);
            if (state.del) await this.delete(ReceiptHandle);
        }
    }

    async send(parsedBody) {
        parsedBody.params.createdAt = new Date();
        return sqs.send(new SendMessageCommand({
            MessageBody: JSON.stringify(parsedBody),
            QueueUrl: this.QueueUrl
        }));
    }
}
```

Listato 5.5: Classe Queue per l'integrazione con AWS SQS

(entità che riceve i messaggi dalla coda) e questo esclude la possibilità di usare un sistema ad eventi. La chiamata di polling crea una connessione che rimane aperta e dormiente sul lato AWS SQS, e quindi se arriva un messaggio sulla coda questo viene inviato subito al consumer, non soltanto alla fine dei 20 secondi. Il consumer non sta girando a vuoto: il suo thread è sospeso in I/O wait e quindi non spreca risorse.

Le code multi-blockchain sono create dinamicamente dalla configurazione mostrata nel listato 5.6. Questo design permette di aggiungere il supporto a una nuova blockchain semplicemente aggiungendo la sua configurazione, senza modificare il codice di gestione delle code.

```
const getOutboundQueues = (queueType) => {
    const queues = {};
    supportedNetworks.forEach(network => {
        queues[network.name] = new Queue(
            getConfigQueueUri(network.name, queueType)
        );
    });
    return queues;
};

const queues = {
    outboundPayedOrders: getOutboundQueues('payed_orders'),
    contractDeployment: getOutboundQueues('contract_deployment'),
    auction: getOutboundQueues('auction'),
    // ... code generiche
};
```

Listato 5.6: Creazione dinamica delle code per ogni blockchain

5.6 Organizzazione degli Endpoint API

Tutte le route dell'API sono versionate sotto il prefisso `/v1` e organizzate in moduli indipendenti, uno per dominio applicativo. Ogni modulo corrisponde a una directory in `src/v1/` e registra un proprio file `routes.ts` che definisce i sotto-percorsi pubblici, protetti e con requisiti di ruolo specifici (come descritto nella sezione 5.1).

Tabella 5.1: Moduli API del backend, raggruppati per area funzionale

Prefisso	**Responsabilità**
Account e identità	
`/user`	Registrazione, login, profilo, indirizzi wallet, 2FA, recupero password, KYC tramite KILT
`/pod-kyc`	Verifica KYC tramite il protocollo Proof of Democracy
Creatori	
`/artist`	Profilo artista, drop e collezioni pubblicati, NFT creati
`/label`	Profilo etichetta, gestione artisti associati, template e drop
Catalogo NFT	
`/template`	CRUD dei template NFT, preferiti, richieste di trasferimento, aste
`/series`	CRUD delle collezioni, assegnazione e spostamento di template
`/drop`	CRUD dei drop, abilitazione al deployment on-chain
`/edition`	Interrogazione di una singola edizione (token) all'interno di un template
`/marketplace`	Ricerca pubblica e navigazione del catalogo (senza autenticazione)
Acquisto e pagamenti	
`/payments`	Creazione e cancellazione ordini, webhook Coinbase e pagamenti on-chain, informazioni USDC
`/stripe`	Creazione e collegamento account Stripe Connect, webhook pagamenti FIAT
`/listing`	Lista early access per template, informazioni gateway disponibili
Mercato secondario	
`/auctions`	Dati aste GBM, generazione firme per offerte, claim e cancellazioni
Piattaforma	
`/history`	Storico trasferimenti e vendite per NFT e template
`/notification`	Lettura e cancellazione notifiche utente
`/newsletter`	Iscrizione alla newsletter della piattaforma
`/redeem`	Trigger per whitelist e airdrop
`/country`	Configurazione della disponibilità dei gateway per paese
`/dashboard`	Pannello operatore: approvazioni, moderazione, gestione royalty
`/root`	Upload file su S3/IPFS, configurazione GBM, DID configuration

La Tabella 5.1 riporta i moduli esposti, raggruppati per area funzionale. Non tutti i moduli sono accessibili allo stesso modo: alcuni endpoint sono pubblici (ad esempio la ricerca nel marketplace o la visualizzazione di un profilo artista), altri richiedono un JWT valido, altri ancora un ruolo specifico come `artistRole`, `labelRole` o `ownerRole`.

5.7 Content Management System

Il Content Management System (CMS) è un servizio separato che condivide i modelli MongoDB con il backend ma espone API dedicate alle operazioni amministrative. Gli unici utenti che hanno accesso a questo servizio sono amministratori di sistema (ruolo admin) e personale interno di Public Pressure (ruolo editor). Le utenze con ruolo editor possono fare azioni relative ad artisti e label, mentre l'utenza con ruolo admin può fare tutte le azioni che sono state previste ed implementate nel servizio, le più importanti sono elencate di seguito:

- **Gestione artisti e label**: creazione, modifica e approvazione dei profili.
- **Gestione template**: creazione e configurazione completa dei template NFT, inclusi upload dei media, configurazione delle vendite e gestione delle royalty.
- **Gestione drop e collezioni**: organizzazione dei template in drop (eventi di lancio) e collezioni tematiche.
- **Gestione whitelist**: creazione e amministrazione delle liste di accesso anticipato.
- **Report e statistiche**: generazione di report sulle vendite, le commissioni e le royalty.
- **Gestione utenti**: operazioni amministrative sugli account utente (ban, verifica manuale, supporto).

Capitolo 6

Smart Contract e Integrazione Blockchain

Questo capitolo analizza in dettaglio gli smart contract sviluppati per Public Pressure, l'implementazione del Diamond Pattern, il meccanismo di asta GBM, il sistema di vendita a prezzo fisso e il servizio di sincronizzazione tra blockchain e database.

6.1 Architettura dei Contratti

L'ecosistema di smart contract di Public Pressure si compone di diverse categorie di contratti che collaborano per gestire l'intero ciclo di vita degli NFT musicali:

- **Diamond Proxy**: il contratto principale che implementa EIP-2535, fungendo da punto di accesso unico per tutte le operazioni, delega le operazioni al contratto Facet appropriato.
- **Facet**: oltre ai contratti standard descritti nel capitolo 2, il Diamond sviluppato per Public Pressure ha altri 2 contratti facet: **GMB** che implementa la logica delle aste, e **TokenSale** il quale gestisce la logica delle vendite a prezzo fisso.
- **ERC721Multiple**: contratto per la creazione e gestione degli NFT, conforme agli standard ERC-721, ERC-2981 ed ERC-721Enumerable.
- **ERC721MultipleLocked**: variante non trasferibile per NFT destinati esclusivamente al mercato primario.
- **Royalties**: contratto per la distribuzione automatica delle royalty tra più destinatari.

6.2 Implementazione del Diamond Pattern (EIP-2535)

Struttura del Diamond

Il Diamond è il contratto principale che riceve tutte le chiamate degli utenti. La sua funzione `fallback()` utilizza `delegatecall` per inoltrare le chiamate al facet appropriato, identificato tramite il selettore della funzione (i primi 4 byte del calldata) come mostrato nel listato 6.1.

```
fallback() external payable {
    LibDiamond.DiamondStorage storage ds;
    bytes32 position = LibDiamond.DIAMOND_STORAGE_POSITION;
    assembly {
        ds.slot := position
    }
    address facet = address(bytes20(ds.facets[msg.sig]));
    require(facet != address(0),
        "Diamond: Function does not exist");
    assembly {
        calldatacopy(0, 0, calldatasize())
        let result := delegatecall(
            gas(), facet, 0, calldatasize(), 0, 0
        )
        returndatacopy(0, 0, returndatasize())
        switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
    }
}
```

Listato 6.1: Funzione fallback del Diamond con delegatecall

La Figura 6.1 illustra il flusso di una chiamata attraverso il Diamond proxy verso un facet (in figura non sono mostrati tutti i facet, ma soltatnto 3 esempi).

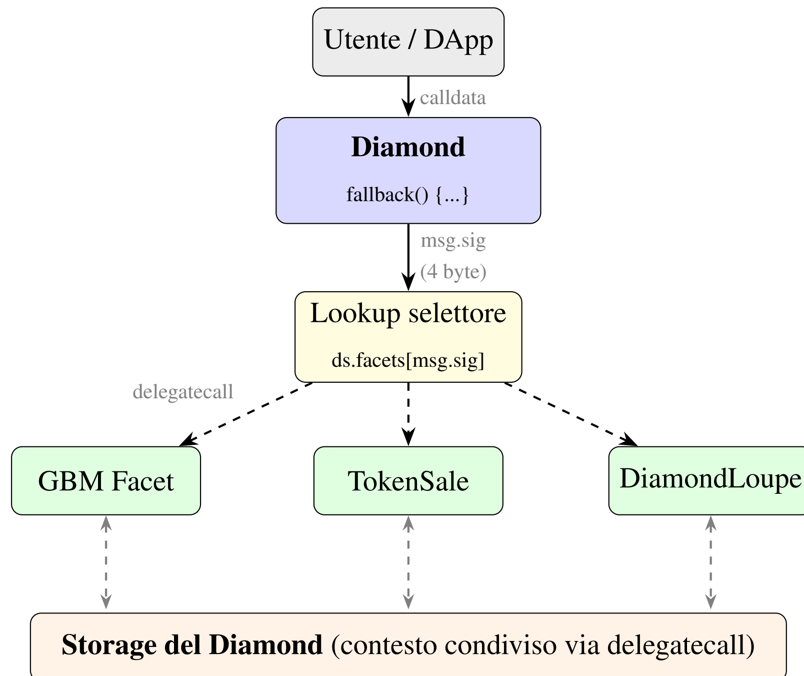

Figura 6.1: Flusso di una chiamata attraverso il Diamond: il selettore della funzione determina il facet di destinazione, che viene eseguito nel contesto di storage del Diamond tramite delegatecall.

Storage Condiviso (AppStorage)

Poiché le `delegatecall` eseguono il codice del facet nel contesto di storage del Diamond, tutti i facet condividono lo stesso spazio di storage. Questo introduce un problema non banale: in Solidity le variabili di stato occupano slot numerati a partire da zero (slot 0, 1, 2, . . .). Se due facet dichiarano variabili di stato proprie (salvate nella memoria persistente), entrambi scriverebbero a partire dallo slot 0, sovrascrivendosi a vicenda.

La soluzione adottata è il pattern **Diamond Storage**: invece di dichiarare variabili di stato normali, tutto lo stato condiviso viene raccolto in un'unica `struct` il cui puntatore viene posizionato in uno slot calcolato come hash di una stringa univoca. Poiché l'output di `keccak256` è distribuito uniformemente su tutto lo spazio degli slot a 256 bit, la probabilità di collisione con gli slot usati dalle variabili locali dei facet è estremamente bassa.

Il Listato 6.2 mostra come viene calcolata e recuperata la posizione dello storage. Le costanti `DIAMOND_STORAGE_POSITION` e `APP_STORAGE_POSITION` sono hash di stringhe descrittive, scelte per essere uniche nell'intero contratto. L'istruzione `assembly { ds.slot := position }` è necessaria perché Solidity non permette di impostare manualmente lo slot di uno storage pointer con sintassi di alto livello: si deve ricorrere

all'assembly inline per fare in modo che ds punti esattamente allo slot desiderato anziché a quello assegnato automaticamente dal compilatore.

In pratica, ogni facet definisce la propria costante univoca. Ad esempio FACET_A_STORAGE_POSITION e FACET_B_STORAGE_POSITION sono calcolate come keccak256 di una stringa diversa per ciascun facet. Il puntatore alla struct di ciascun facet viene quindi posizionato nello slot corrispondente alla propria costante, garantendo che ogni facet scriva e legga esclusivamente nella propria area di storage, senza interferire con gli altri. Non esiste quindi un'unica area condivisa, ma tante aree isolate quanti sono i facet, ciascuna identificata da una posizione pseudo-casuale e univoca nello spazio degli slot.

```
bytes32 constant DIAMOND_STORAGE_POSITION =
keccak256("diamond.standard.diamond.storage");
bytes32 constant APP_STORAGE_POSITION =
keccak256("app.storage");

function diamondStorage() internal pure
returns (DiamondStorage storage ds) {
  bytes32 position = DIAMOND_STORAGE_POSITION;
  assembly {
    ds.slot := position
  }
}
```

Listato 6.2: Posizionamento deterministico dello storage tramite hash di stringa univoca

Tutta la logica applicativa condivisa tra i facet (aste, vendite, configurazioni) è raccolta in un'unica struttura AppStorage, recuperata con lo stesso meccanismo. Centralizzare lo stato in un'unica struct garantisce che tutti i facet leggano e scrivano gli stessi dati senza conflitti, ed elimina la necessità di sincronizzare variabili distribuite tra contratti diversi. Il Listato 6.3 mostra un estratto dei campi principali.

Processo di Deploy del Diamond

Il deploy del Diamond avviene in più fasi:

1. **Deploy del DiamondCutFacet**: il facet che gestisce le operazioni di aggiornamento del Diamond.

2. **Deploy del Diamond**: il contratto proxy principale, configurato con l'indirizzo del DiamondCutFacet e del proprietario (un wallet gestito da Public Pressure).

3. **Deploy del DiamondInit**: contratto di inizializzazione che esegue la configurazione iniziale dello storage.

```
struct AppStorage {
  // Mapping da auctionID ai dati dell'asta
  mapping(uint256 => Auction) auctions;
  // Mapping da auctionID alla rappresentazione del token
  mapping(uint256 => token_representation) tokenMapping;
  // Mapping multi-livello per le aste
  mapping(address => mapping(bytes4 =>
  mapping(uint256 => mapping(uint256 => uint256))))
  auctionMapping;
  // Configurazione delle collezioni
  mapping(address => collection_config) collections;
  // Commissioni della piattaforma
  address PPFeeAddress;
  uint256 PPFeePMPercentage;  // Mercato primario
  uint256 PPFeeSMPercentage;  // Mercato secondario
  // ...
}
```

Listato 6.3: Estratto di AppStorage: lo stato condiviso tra tutti i facet del Diamond

4. **Deploy dei facet**: compilazione e deploy di DiamondLoupeFacet, OwnershipFacet, GBM e TokenSale.

5. **Diamond Cut**: operazione che registra tutti i selettori delle funzioni dei facet nel Diamond, associando ogni selettore all'indirizzo del facet corrispondente.

6.3 Contratto GBM: Meccanismo di Asta Gamificato

Il contratto GBM implementa il meccanismo di asta gamificato come facet del Diamond. Le funzionalità principali sono descritte di seguito.

Funzione bid()

La funzione `bid()` gestisce il piazzamento delle offerte con le verifiche e operazioni mostrate nel listato 6.4.

Calcolo degli Incentivi

La funzione `calculateIncentives()`, mostrata nel listato 6.5, determina l'incentivo dovuto al prossimo offerente che verrà superato. L'incentivo cresce proporzionalmente all'eccesso dell'offerta rispetto all'incremento minimo richiesto. Il limite massimo (`incMax`) previene situazioni in cui un'offerta molto elevata genererebbe incentivi eccessivi, proteggendo il venditore da perdite.

```
function bid(uint256 _auctionID, uint256 _bidAmount,
    uint256 _highestBid) external payable override {

    require(s.tokenMapping[_auctionID].contractAddress
        != address(0x0), "auctionID does not exist");
    require(s.collections[s.tokenMapping[_auctionID]
        .contractAddress].biddingAllowed,
        "bidding is currently not allowed");
    require(_bidAmount > getAuctionMinimumBid(_auctionID),
        "cannot be less than minimum bid");
    require(_highestBid == s.auctions[_auctionID].highestBid,
        "highest bid mismatch");
    require(getAuctionStartTime(_auctionID) <= block.timestamp,
        "Auction has not started yet");
    require(getAuctionEndTime(_auctionID) >= block.timestamp,
        "Auction has already ended");
    require(msg.value == _bidAmount,
        "bid amount doesn't match currency sent");

    // Estensione Hammer Time
    if (getAuctionEndTime(_auctionID) <
        block.timestamp + getHammerTimeDuration(_auctionID)) {
        s.auctions[_auctionID].endTime =
            block.timestamp + getHammerTimeDuration(_auctionID);
        emit Auction_EndTimeUpdated(_auctionID,
            s.auctions[_auctionID].endTime);
    }

    // Calcolo incentivi per il nuovo offerente
    s.auctions[_auctionID].dueIncentives =
        calculateIncentives(_auctionID, _bidAmount);

    // Aggiornamento offerta massima
    s.auctions[_auctionID].highestBidder = msg.sender;
    s.auctions[_auctionID].highestBid = _bidAmount;

    // Rimborso offerente precedente + incentivi
    if ((previousHighestBid + duePay) != 0) {
        (bool sent, ) = previousHighestBidder
            .call{value: previousHighestBid + duePay}("");
        require(sent, "Failed to refund ETH");
    }
}
```

Listato 6.4: Estratto della funzione bid() del contratto GBM

```
function calculateIncentives(uint256 _auctionID,
    uint256 _newBidValue) internal view returns (uint256) {

    uint256 bidDecimals = getAuctionBidDecimals(_auctionID);
    uint256 bidIncMax = getAuctionIncMax(_auctionID);

    uint256 baseBid = s.auctions[_auctionID].highestBid
        * (bidDecimals + getAuctionStepMin(_auctionID))
        / bidDecimals;

    if (baseBid == 0) baseBid = 1;

    uint256 decimaledRatio =
        ((bidDecimals * getAuctionBidMultiplier(_auctionID)
          * (_newBidValue - baseBid)) / baseBid)
        + getAuctionIncMin(_auctionID) * bidDecimals;

    // Limite massimo all'incentivo massimo
    if (decimaledRatio > (bidDecimals * bidIncMax)) {
        decimaledRatio = bidDecimals * bidIncMax;
    }

    return (_newBidValue * decimaledRatio)
        / (bidDecimals * bidDecimals);
}
```

Listato 6.5: Calcolo degli incentivi GBM

Funzione claim()

La funzione `claim()` è chiamata dopo la fine dell'asta e del periodo di grazia per finalizzare la vendita:

1. Verifica che l'asta sia terminata e che il periodo di cancellazione sia trascorso.
2. Calcola le commissioni della piattaforma (percentuale differenziata tra mercato primario e secondario).
3. Se il contratto NFT supporta ERC-2981, interroga `royaltyInfo()` e distribuisce le royalty.
4. Trasferisce i proventi rimanenti al beneficiario (venditore).
5. Trasferisce l'NFT al vincitore dell'asta.

La distinzione tra mercato primario e secondario è gestita dal flag `primaryMarketAuction`: nel mercato primario le commissioni della piattaforma sono calcolate con `PPFeePMPercentage`, nel secondario con `PPFeeSMPercentage`. Le royalty ERC-2981 vengono distribuite solo nel mercato secondario, in quanto nel primario l'intero ricavato va all'artista/creatore.

Cancellazione dell'Asta

Il venditore può cancellare un'asta durante il periodo di grazia tramite la funzione `cancelAuction()`, pagando come penale l'importo degli incentivi dovuti e accumulati. Questo meccanismo scoraggia la cancellazione opportunistica delle aste (ad esempio, annullare un'asta il cui prezzo finale è inferiore alle aspettative) mantenendo al contempo la possibilità di gestire situazioni eccezionali.

6.4 Contratto TokenSale: Vendita a Prezzo Fisso

Il contratto `TokenSale` gestisce le vendite a prezzo fisso sul mercato secondario. Un utente che possiede un NFT può metterlo in vendita specificando un prezzo, e qualsiasi altro utente può acquistarlo pagando il prezzo richiesto.

La funzione `buyToken()` esegue atomicamente:

- Verifica che il pagamento corrisponda al prezzo richiesto.
- Deduce le commissioni della piattaforma.
- Interroga ERC-2981 per le royalty e le distribuisce al destinatario.
- Trasferisce i proventi residui al venditore.
- Trasferisce l'NFT al compratore.

6.5 Contratto ERC721Multiple

Ogni template pubblicato su Public Pressure viene distribuito come un'istanza separata del contratto `ERC721Multiple`: è questo contratto che detiene la proprietà degli NFT, tiene traccia di chi possiede ogni edizione e ne gestisce i trasferimenti. Il Diamond (con i facet GBM e TokenSale) non possiede i token bensì li *gestisce* tramite l'approvazione che il contratto ERC721Multiple concede all'indirizzo del Diamond attraverso `setApprovalForAll`, che lo autorizza a trasferire token per conto dei proprietari.

Stack di standard implementati

Il contratto estende una serie di classi OpenZeppelin Upgradeable, ciascuna con uno scopo preciso. `ERC721Upgradeable` fornisce l'implementazione base dello standard ERC-721: assegnazione, trasferimento e verifica della proprietà dei token. `ERC2981Upgradeable` aggiunge il supporto allo standard di royalty: ogni volta che un marketplace

on-chain (incluso il Diamond di Public Pressure) esegue una vendita, può interrogare il contratto per sapere a chi versare la percentuale di royalty e in quale misura. `ERC721EnumerableUpgradeable` permette di enumerare tutti i token della collezione e tutti i token posseduti da un determinato indirizzo (wallet), funzionalità utile per costruire la vista della collezione di un utente. `PausableUpgradeable` e `OwnableUpgradeable` forniscono rispettivamente la possibilità di congelare i trasferimenti in caso di emergenza e un meccanismo di accesso riservato al proprietario del contratto. Infine, `UUPSUpgradeable` implementa il pattern di aggiornabilità UUPS, che permette di sostituire la logica del contratto preservando lo storage e l'indirizzo — lo stesso principio del Diamond, ma applicato al singolo contratto NFT.

Numerazione delle edizioni e offset multi-chain

Quando viene fatto il deploy di un template, il contratto viene inizializzato con due parametri fondamentali: `totalSupply_` (il numero di edizioni disponibili) e `offset_` (un intero che indica su quale blockchain si sta effettuando il deploy, a partire da zero). Questi due valori determinano l'intervallo di token ID di cui questo deploy potrà fare il minting.

```
1  _tokenIdCounter = CountersUpgradeable.Counter(
2  totalSupply_ * offset_
3  );
4  _limit = totalSupply_ * (offset_ + 1);
```

Listato 6.6: Calcolo dell'intervallo di token ID in base a supply e offset

Il Listato 6.6 mostra il motivo per cui l'offset esiste. Public Pressure supporta quattro blockchain in produzione: lo stesso template viene distribuito su ciascuna di esse con un contratto ERC721Multiple distinto, uno per chain. Se tutti i contratti iniziassero a numerare i token da 1, il token ID 1 esisterebbe su Moonbeam, su Polygon, su Base e su Astar contemporaneamente, rendendo impossibile per il backend identificare univocamente a quale edizione fisica si riferisce un evento blockchain.

L'offset risolve il problema assegnando a ciascun deploy un intervallo non sovrapposto. Per un template con 100 edizioni:

Blockchain	offset	Token ID
Moonbeam	0	1 – 100
Polygon	1	101 – 200
Base	2	201 – 300
Astar	3	301 – 400

Quando il servizio di blockchain-pulling riceve un evento di trasferimento con token ID 243, sa immediatamente che si tratta dell'edizione 43 distribuita su Base — senza dover interrogare ogni chain per disambiguare.

Variante ERC721MultipleLocked

`ERC721MultipleLocked` è una variante del contratto in cui le funzioni di trasferimento `transferFrom` e `safeTransferFrom` vengono sovrascritte con un `require(false)`, rendendo ogni token non trasferibile dopo il mint. Questa variante viene usata per le edizioni destinate esclusivamente al mercato primario: l'acquirente riceve il token come prova permanente di acquisto, ma non può rivenderlo su nessun marketplace.

6.6 Sistema di Royalty

Il sistema di royalty di Public Pressure opera su due livelli:

Royalty On-Chain (ERC-2981)

Il contratto `ERC721Multiple` implementa ERC-2981, restituendo l'indirizzo del contratto `Royalties` e la percentuale configurata. I contratti GBM e TokenSale verificano il supporto a ERC-2981 tramite ERC-165 (`supportsInterface(0x2a55205a)`) e, se supportato, distribuiscono automaticamente le royalty durante le transazioni on-chain.

Royalty Off-Chain (Pagamenti FIAT)

Per gli acquisti effettuati tramite Stripe, le royalty vengono gestite a livello applicativo:

1. Il webhook di Stripe notifica il backend del pagamento avvenuto.
2. Il backend calcola le commissioni della piattaforma e l'importo netto.
3. Per ogni destinatario di royalty nel template, il backend crea un trasferimento Stripe verso l'account dell'artista/label.
4. Viene creato un record `Accountability` per tracciare ogni pagamento.

Questo sistema duale garantisce che gli artisti ricevano le royalty indipendentemente dal metodo di pagamento utilizzato dall'acquirente.

Il listato 6.7 mostra la struttura del contratto `ERC721Multiple` e la funzione di inizializzazione, che configura tutti gli standard implementati.

```
contract ERC721Multiple is
    Initializable,
    ERC721Upgradeable,
    ERC2981Upgradeable,
    ERC721EnumerableUpgradeable,
    PausableUpgradeable,
    OwnableUpgradeable,
    UUPSUpgradeable
{
    using CountersUpgradeable
        for CountersUpgradeable.Counter;
    CountersUpgradeable.Counter private _tokenIdCounter;
    uint256 private _limit;
    address private _gbmAddress;
    string private _contractBaseURI;

    function initialize(
        string memory name_, string memory symbol_,
        uint256 totalSupply_, uint256 offset_,
        string memory baseURI_, address gbmAddress_,
        address upgraderAddress_,
        uint96 royaltiesPercent_,
        address addressSplit_
    ) public initializer {
        __ERC721_init(name_, symbol_);
        __ERC2981_init();
        __ERC721Enumerable_init();
        __Pausable_init();
        __Ownable_init();
        __UUPSUpgradeable_init();
        _tokenIdCounter = CountersUpgradeable.Counter(
            totalSupply_ * offset_);
        _limit = totalSupply_ * (offset_ + 1);
        _contractBaseURI = baseURI_;
        _gbmAddress = gbmAddress_;
    }
}
```

Listato 6.7: Contratto ERC721Multiple con stack OpenZeppelin Upgradeable

6.7 Servizio di Blockchain-Pulling

Il servizio di blockchain-pulling è un componente critico che mantiene sincronizzato il database applicativo con lo stato delle blockchain. Per ogni blockchain supportata, un'istanza separata del servizio esegue un loop continuo di monitoraggio.

Logica di Pulling

Il servizio implementa il seguente algoritmo:

1. All'avvio, recupera dal database il numero dell'ultimo blocco processato.
2. In un loop infinito:
   (a) Interroga il nodo blockchain per l'ultimo blocco confermato.
   (b) Se ci sono blocchi da processare:
      - Se il gap è grande (più di 3 blocchi, o 30 per Base), processa i blocchi parallelamente ed in gruppi per recuperare velocemente.
      - Se il gap è piccolo, processa i blocchi uno alla volta.

   (c) Per ogni blocco, recupera tutte le transazioni ed i rispettivi receipt (contenenti gli esiti delle transazioni).
   (d) Filtra i log per eventi rilevanti ovvero che riguardano i wallet degli utenti della piattaforma oppure dei contratti di proprietà della piattaforma.
   (e) Processa ogni evento all'interno di una transazione MongoDB per garantire atomicità.
3. Attende 10 secondi prima di ripetere il controllo.

Processamento degli Eventi Rilevati

Gli eventi recuperati dalla blockchain vengono gestiti dal backend in base al tipo di evento, aggiornando la base di dati. Un aspetto critico è la gestione della collisione tra le firme degli eventi: la funzione `Transfer(address,address,uint256)` di ERC-721 ha lo stesso hash keccak256 della funzione `Transfer` del contratto USDC (ERC-20). Il processore distingue i due casi verificando l'indirizzo del contratto emittente come mostrato nel listato 6.8.

Gli eventi processati producono diverse azioni sul database:

- **Transfer (ERC-721)**: creazione di un record `NFTHistory`, aggiornamento del proprietario dell'edizione, aggiornamento dello stato della transazione associata.

```
if (contract === 'ERC721Multiple' &&
    elem.address.toLowerCase() === usdcAddress.toLowerCase())
    return null;  // Ignora i trasferimenti USDC
```

Listato 6.8: Gestione della collisione di firme tra ERC-721 e USDC

- **Auction_Initialized**: creazione di un record `Auction` con i metadati dell'asta (tempi, parametri, venditore).
- **Auction_BidPlaced**: creazione di un record `BidHistory`, aggiornamento dell'offerta corrente nell'asta, invio di notifiche al creatore dell'asta ed all'utente appena superato.
- **Auction_BidRemoved**: aggiornamento dello stato dell'offerta precedente come "outbid".
- **AuctionClaimed**: finalizzazione dell'asta nel database, creazione dello storico del trasferimento.
- **TokenSold**: aggiornamento dello stato della vendita, creazione dello storico della transazione.

Gestione dell'attesa di Conferma dei Blocchi

Come discusso nel Capitolo 2, i blocchi più recenti di una blockchain possono essere soggetti a riorganizzazioni. Il servizio di blockchain-pulling affronta questo problema processando solo i blocchi già **confermati**, ovvero blocchi sufficientemente profondi nella catena da essere considerati irreversibili.

Tuttavia, questo approccio introduce una latenza tra l'evento on-chain e la sua visibilità sulla piattaforma. Per bilanciare sicurezza e reattività, il sistema adotta la strategia di mostrare all'utente lo stato dell'ordine come "in elaborazione" immediatamente dopo il pagamento, prima ancora che la creazione avvenga sulla blockchain e che venga rilevata dal servizio di pulling. L'utente percepisce così una risposta rapida, mentre il sistema attende la conferma definitiva dalla blockchain prima di considerare l'ordine completato.

Il numero di blocchi che devono essere creati prima che un certo blocco sia considerato irreversibile varia in base alla blockchain specifica: Base, con tempi di blocco più rapidi, utilizza 30 blocchi, mentre le altre reti implementate sulla piattaforma utilizzano 3 blocchi.

Registro degli Eventi

Il servizio mantiene un registro (`BlockLogs`) di tutti i blocchi processati, permettendo di riprendere il pulling dal punto esatto in caso di riavvio del servizio. Questo garantisce che nessun evento venga perso anche in caso di crash o aggiornamento del servizio.

La Figura 6.2 riassume la pipeline di processamento degli eventi del servizio di blockchain-pulling.

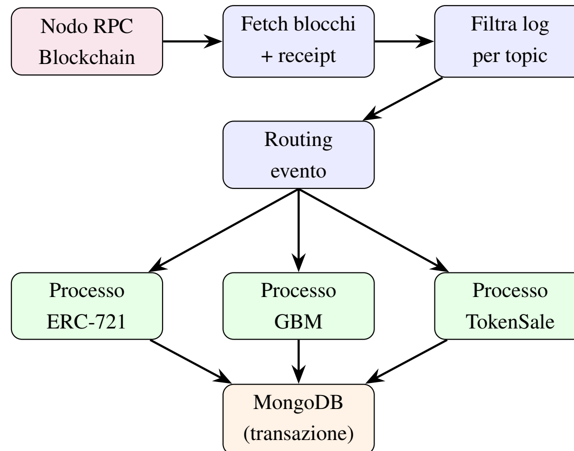

Figura 6.2: Pipeline di processamento del servizio di blockchain-pulling: i blocchi vengono recuperati dal nodo RPC, gli eventi vengono filtrati e instradati ai processori specializzati, che aggiornano il database in modo atomico.

Capitolo 7

Gestione dei Pagamenti

Questo capitolo esamina in dettaglio le soluzioni implementate per la gestione dei pagamenti in Public Pressure, con particolare attenzione al problema della concorrenza tra metodi di pagamento diversi.

7.1 Panoramica dei Gateway di Pagamento

Public Pressure supporta tre gateway di pagamento (più il riscatto di NFT in modo gratuito), elencati nella tabella 7.1, ciascuno con caratteristiche, tempi di conferma e complessità differenti.

Tabella 7.1: Confronto dei gateway di pagamento supportati

Gateway	Descrizione	Tempo conferma	Tipo
`stripe`	Carta di credito/debito via Stripe	2–5 secondi	FIAT
`cc`	Criptovaluta via Coinbase Commerce	1–15 minuti	Crypto
`erc20`	Pagamento diretto in USDC on-chain	30s–5 minuti	Crypto
`free`	Claim gratuito (airdrop, whitelist)	Immediato	N/A

La coesistenza di questi gateway è una delle caratteristiche distintive di Public Pressure, ma introduce complessità significative nella gestione della concorrenza e della coerenza dei dati. La Figura 7.1 confronta i flussi dei tre gateway principali. Questi differiscono non solo per il mezzo di pagamento, ma per la posizione che il backend occupa nel flusso e per chi detiene la custodia dei fondi in ogni momento.

Con **Stripe** il backend è l'intermediario attivo: crea la sessione di pagamento, Stripe addebita la carta dell'utente e trattiene i fondi sul proprio sistema, poi notifica il backend tramite webhook. I fondi non toccano mai la blockchain — restano nel circuito bancario

fino al successivo payout verso l'account Stripe Connect dell'artista. È il metodo più accessibile per chi non ha familiarità con le criptovalute, ma dipende interamente da un servizio centralizzato di terze parti.

Con **Coinbase Commerce** il flusso è simile dal punto di vista architetturale: il backend crea un charge su Coinbase, l'utente invia criptovaluta all'indirizzo di deposito fornito da Coinbase, ed il backend riceve un webhook di conferma. La differenza è che il pagamento transita sulla blockchain, ma è Coinbase a fare da custode: l'utente non interagisce direttamente con gli smart contract della piattaforma, e la conversione in FIAT avviene internamente a Coinbase. Rispetto a Stripe, i tempi di conferma sono più lunghi e variabili (da uno a quindici minuti), poichè dipendono dal tempo di conferma dei blocchi sulla rete scelta dall'utente.

Con il pagamento **ERC-20 on-chain** il backend esce dal flusso del denaro: il trasferimento di USDC avviene direttamente tra il wallet dell'utente e lo smart contract, senza intermediari che custodiscano i fondi. Il backend non vede mai i token — si limita a ricevere una notifica quando la transazione è confermata sulla blockchain e ad aggiornare lo stato dell'ordine di conseguenza. È il metodo più trasparente e coerente con i principi della decentralizzazione, ma richiede che l'utente possieda una wallet e sappia usarlo, e che abbia già autorizzato il contratto a prelevare i propri token tramite una transazione di `approve`.

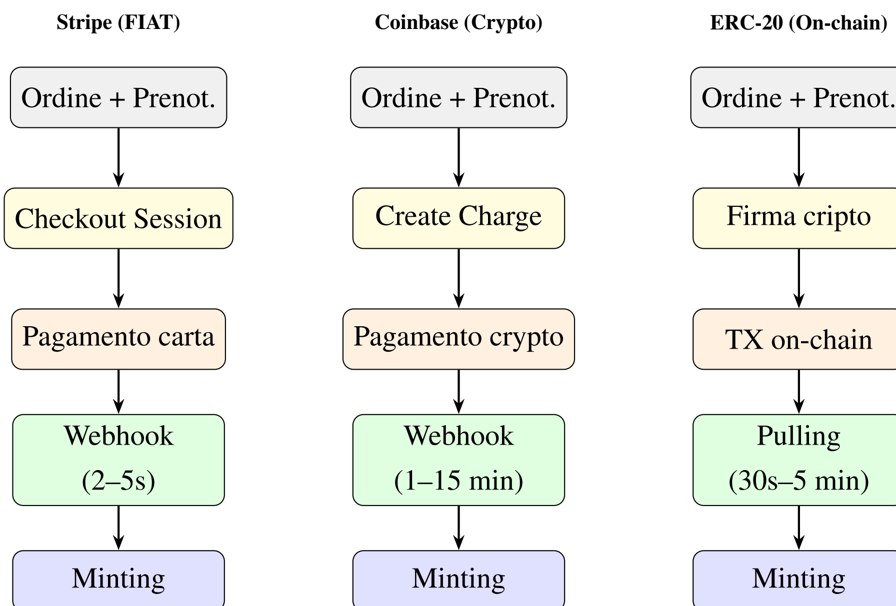

Figura 7.1: Confronto dei flussi di pagamento per i tre gateway principali. I tempi di conferma variano significativamente tra FIAT e crypto.

7.2 Template Curated

Per alcuni artisti o etichette, gestire autonomamente gli incassi e le distribuzioni di royalty tramite Stripe Connect o Coinbase Commerce può risultare troppo oneroso dal punto di vista amministrativo: richiede la creazione e verifica di account su piattaforme terze, la gestione dei dati bancari, e la riconciliazione mensile dei pagamenti ricevuti. Public Pressure offre in questi casi una modalità alternativa chiamata **curated**.

In un template curated, **Public Pressure agisce da intermediario commerciale completo**: raccoglie tutti i pagamenti per conto dell'artista e provvede a liquidare le royalty manualmente, tramite bonifico bancario, al termine di ogni mese. L'artista non necessita di nessun account Stripe Connect né di configurazioni wallet, e la piattaforma si occupa di tutta la parte finanziaria.

Questa scelta produce conseguenze dirette sul comportamento del sistema:

- **Gateway disponibili**: il gateway Coinbase Commerce viene disabilitato automaticamente per i template curated. Poiché Coinbase non supporta la distribuzione automatica delle royalty verso un intermediario, la modalità curated è compatibile solo con Stripe (FIAT) e con il pagamento diretto in ERC-20 on-chain.
- **Nessun trasferimento Stripe immediato**: a differenza del flusso standard, in cui il backend effettua trasferimenti Stripe verso gli account degli artisti al momento della vendita, per i template curated questi trasferimenti vengono omessi. Il pagamento viene incassato sull'account principale di Public Pressure senza associarlo ad un `transfer_group`.
- **Accountability con liquidazione differita**: vengono comunque creati i documenti `Accountability` per ogni royalty dovuta, ma con il campo `payed` impostato a `false`. Questi record costituiscono il debito della piattaforma verso gli artisti e vengono saldati al termine del ciclo mensile di rendicontazione, al di fuori del sistema di pagamento automatico.

La modalità curated non è un'opzione che l'acquirente vede o sceglie: è una configurazione del template, stabilita contrattualmente tra Public Pressure e l'artista o l'etichetta al momento della creazione. Dal punto di vista dell'acquirente il flusso di acquisto rimane invariato; è il backend a comportarsi diversamente nella fase di distribuzione dei proventi.

7.3 Integrazione con Stripe

Configurazione degli Account

Stripe è utilizzato sia per ricevere i pagamenti dagli acquirenti sia per distribuire i proventi agli artisti. Il sistema utilizza **Stripe Connect** con account di tipo **Express**, che permette agli artisti di configurare il proprio account Stripe senza che Public Pressure debba gestire direttamente i dati finanziari sensibili.

Il flusso di onboarding di un artista su Stripe prevede:

1. L'artista richiede l'attivazione dei pagamenti tramite il CMS.
2. Il backend crea un account Express su Stripe con le capability `transfers` e `card_payments`.
3. Viene generato un link di onboarding che reindirizza l'artista alla pagina Stripe per completare la verifica dell'identità e la configurazione del conto bancario.
4. Al completamento, lo stato `stripeVerified` dell'artista viene aggiornato.

Flusso di Pagamento

Quando un utente sceglie di pagare con carta:

1. Il backend crea una **Checkout Session** di Stripe con il prezzo in USD, il nome del template e il metodo di pagamento `card`.
2. La sessione utilizza `capture_method:"manual"`, che autorizza il pagamento senza addebitarlo immediatamente. Questo permette di catturare il pagamento solo dopo aver confermato la disponibilità dell'NFT.
3. L'utente viene reindirizzato alla pagina di checkout ospitata da Stripe, dove inserisce i dati della carta.
4. Al completamento del pagamento, Stripe invia un webhook `payment_intent.succeeded` al backend.

Gestione del Webhook

Il webhook di Stripe è uno dei componenti più critici del sistema. Alla ricezione dell'evento `payment_intent.succeeded`:

1. Il backend verifica la firma del webhook utilizzando il segreto condiviso con Stripe e il body raw della richiesta.

2. Recupera i dettagli della transazione, incluso l'importo netto dopo le commissioni di Stripe.

3. Calcola la commissione della piattaforma (PP fee) come percentuale dell'importo netto.

4. Crea un documento `FiatSale` che registra l'importo lordo, l'importo netto dopo le commissioni del gateway e l'importo dopo le commissioni della piattaforma.

5. Per ogni destinatario di royalty nel template, crea un trasferimento Stripe:

```
await stripeInstance.transfers.create({
    amount: royalty.amount,
    currency: 'usd',
    destination: royalty.accountId,
    source_transaction: chargeId
});
```

Listato 7.1: Creazione dei trasferimenti di royalty su Stripe

6. Crea i documenti `Accountability` per tracciare ogni pagamento di royalty.

7. Aggiorna l'ordine a stato `payed` e invia il messaggio sulla coda per il minting.

7.4 Integrazione con Coinbase Commerce

Coinbase Commerce permette agli utenti di pagare in criptovaluta tramite un'interfaccia ospitata da Coinbase. Il flusso è simile a quello di Stripe:

1. Il backend crea un "charge" su Coinbase Commerce specificando il prezzo in USD e i metadati dell'ordine.

2. L'utente viene reindirizzato alla pagina di pagamento di Coinbase, dove può scegliere la criptovaluta con cui pagare (Bitcoin, Ethereum, USDC, ecc.).

3. Coinbase gestisce la conversione e la conferma del pagamento, poi invia un webhook al backend.

4. Il backend processa il webhook in modo analogo a Stripe.

7.5 Pagamenti Diretti in ERC-20 (USDC)

Il gateway ERC-20 rappresenta il metodo di pagamento più "nativo" della blockchain, permettendo agli utenti di pagare direttamente in USDC (stablecoin ancorata al dollaro) sulla blockchain scelta.

Flusso di Pagamento

1. Il backend contatta un servizio esterno di gestione dei pagamenti crypto, inviando il valore in centesimi USD e i metadati dell'ordine.
2. Il servizio esterno genera una **firma crittografica** che autorizza il contratto a ricevere il pagamento per quell'ordine specifico.
3. Il backend restituisce al frontend la firma, l'indirizzo del contratto di pagamento, l'importo e l'indirizzo della valuta (USDC) sulla blockchain selezionata.
4. Il frontend presenta la transazione all'utente, che la approva tramite il proprio wallet (MetaMask, WalletConnect, ecc.).
5. La transazione viene inviata alla blockchain e, una volta confermata, il servizio di pagamenti crypto invia un webhook al backend.

Verifica del Webhook

La verifica del webhook dei pagamenti crypto utilizza un meccanismo basato su certificati digitali, simile a quello di AWS SNS:

```
const verified = crypto.verify(
    'sha256WithRSAEncryption',
    Buffer.from(stringToSign, 'utf-8'),
    publicKey,
    signature
);
```

Listato 7.2: Verifica della firma del webhook crypto

Questo meccanismo garantisce che solo il servizio di pagamenti autorizzato possa notificare il backend di un pagamento avvenuto.

7.6 Il Problema della Concorrenza

La concorrenza tra pagamenti è il problema più complesso affrontato nell'implementazione. Il problema può essere formalizzato come segue:

Dato un NFT con N edizioni disponibili, e M utenti che tentano di acquistarlo contemporaneamente con metodi di pagamento diversi (ciascuno con tempi di conferma differenti), garantire che vengano vendute esattamente N edizioni e che nessun utente paghi per un'edizione non disponibile.

Scenari Problematici

1. **Race condition Stripe vs. Crypto**: l'utente A inizia un pagamento con carta (conferma in 5 secondi), l'utente B inizia un pagamento in USDC (conferma in 2 minuti). Se rimane una sola edizione, il sistema deve garantire che solo uno dei due ottenga l'NFT.

2. **Transazioni non completate**: l'utente B avvia un pagamento crypto ma non completa la transazione. L'edizione resta "bloccata" dalla prenotazione, impedendo ad altri utenti di acquistarla.

Race condition Stripe vs. Crypto

La Figura 7.2 illustra il problema della race condition e la soluzione tramite prenotazione atomica.

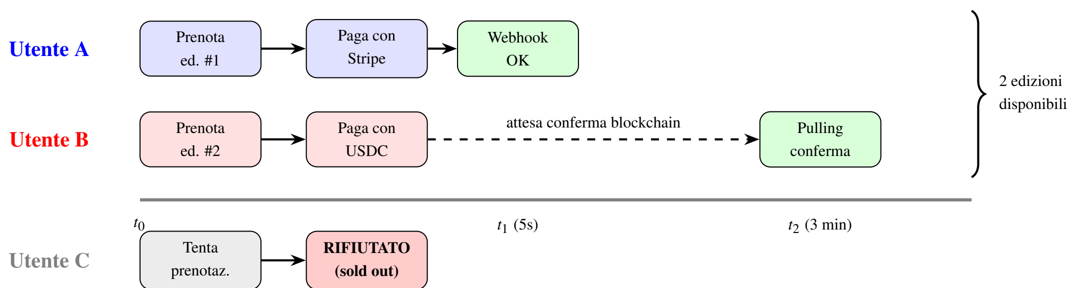

Figura 7.2: Gestione della concorrenza: la prenotazione atomica al tempo t_0 garantisce che ogni edizione sia assegnata a un solo utente, indipendentemente dai tempi di conferma del pagamento.

Soluzione Implementata

La soluzione adottata si basa su tre meccanismi coordinati:

Prenotazione Atomica

Il sistema di prenotazione utilizza operazioni atomiche di MongoDB per garantire che il numero di edizioni prenotate non superi mai il numero di edizioni disponibili.

L'operazione `findOneAndUpdate` con la condizione `$expr` è atomica e thread-safe a livello del database engine, eliminando le race condition a livello applicativo.

Se la prenotazione fallisce (edizioni esaurite), l'ordine non viene creato e l'utente riceve un errore immediato, prima ancora di iniziare il pagamento.

Transazioni MongoDB

Tutte le operazioni critiche (creazione ordine, prenotazione, aggiornamento stato) avvengono all'interno di transazioni MongoDB con isolamento a livello di snapshot. Questo garantisce che:

- Se la creazione dell'ordine fallisce, la prenotazione viene automaticamente annullata (rollback).
- Due ordini concorrenti per le ultime edizioni vedono uno snapshot coerente del database.
- Non è possibile creare un ordine senza la corrispondente prenotazione.

Transazioni non completate

Per gestire il caso di pagamenti non completati (specialmente rilevante per i pagamenti crypto, che possono richiedere diversi minuti), il sistema implementa:

- Un **timeout** sugli ordini: se il pagamento non viene confermato entro un periodo prestabilito, l'ordine viene automaticamente cancellato e le edizioni liberate. Il valore del timeout è impostato in base al gateway, ed eventualmente la blockchain, scelti dall'utente in modo da rendere impossibile la conferma del pagamento dopo il timeout.
- La **cancellazione esplicita**: l'utente può annullare l'ordine in qualsiasi momento prima della conferma del pagamento, liberando immediatamente le edizioni.
- La gestione degli ordini **falliti**: se il webhook di un gateway segnala un pagamento fallito o scaduto, l'ordine viene marcato come `failed` e le edizioni vengono liberate.

Compromessi e Limitazioni

La soluzione adottata rappresenta un compromesso tra sicurezza e esperienza utente:

- **Blocco temporaneo delle edizioni**: durante il periodo di pagamento (che per i pagamenti crypto può anche arrivare ad alcuni minuti), le edizioni prenotate non sono disponibili per altri utenti. Questo può creare una percezione di scarsità artificiale, ma è necessario per evitare vendite doppie. Esiste il rischio che un utente (oppure un gruppo) possa eseguire un attacco di tipo DDoS iniziando il flusso di acquisto senza mai concludere, mantenendo effettivamente bloccato un prodotto. Questo rischio esiste ma è molto basso dato che il tempo limite per completare l'acquisto non è costante, ma cambia in base al metodo di pagamento e blockchain scelta. Un'ulteriore protezione potrebbe essere l'aggiunta un valore casuale al calcolo del timeout per rendererlo imprevedibile ed evitare questo tipo di attacco.

- **Latenza per i pagamenti crypto**: un utente che paga in criptovaluta deve attendere la finalizzazione della transazione prima di vedere l'NFT nel proprio wallet. Il sistema mitiga questa latenza mostrando immediatamente lo stato "in elaborazione".

- **Rischio residuo di riorganizzazione**: sebbene il servizio di pulling processi solo blocchi confermati, esiste un rischio teorico (estremamente basso) di riorganizzazione profonda che potrebbe invalidare una transazione già confermata dal gateway crypto.

Questi compromessi sono stati discussi e accettati dal team di sviluppo, privilegiando la sicurezza (nessuna doppia vendita) rispetto alla reattività immediata. In un contesto di NFT, dove ogni token è unico e non sostituibile, la prevenzione delle doppie vendite è un requisito non negoziabile.

7.7 Disponibilità dei Gateway per Paese

Non tutti i gateway sono disponibili in ogni paese. La restrizione si basa sul paese di residenza dell'**artista o etichetta** che ha pubblicato il template: se un artista risiede in un paese in cui Coinbase Commerce non è consentito, nessun acquirente potrà usare quel gateway per quel template, indipendentemente da dove si trova.

Il sistema recupera la configurazione del paese del creator al momento della creazione dell'ordine e verifica la compatibilità con il gateway richiesto:

Il paese dell'acquirente viene invece utilizzato in una fase precedente, durante la verifica dell'identità (KYC): al momento della conferma del numero di telefono, l'utente dichiara il proprio paese di residenza, che il sistema valida confrontandolo con

```
const creator = await model.findById(template.createdBy._id).
  lean();
const countryConfigs = await Country.findOne({ code: creator.
  country }).lean();

if (gateway === 'stripe' && !countryConfigs.stripe)
return reply.badRequest(
'RESERVATION_ERROR.CANNOT_BUY_WITH_STRIPE');
if (gateway === 'cc' && !countryConfigs.coinbase)
return reply.badRequest(
'RESERVATION_ERROR.CANNOT_BUY_WITH_CC');
if (gateway === 'erc20' && !countryConfigs.erc20)
return reply.badRequest(
'RESERVATION_ERROR.CANNOT_BUY_WITH_ERC20');
```

Listato 7.3: Verifica della disponibilità del gateway in base al paese del creator

il prefisso internazionale del numero fornito. Questa informazione viene poi usata per bloccare l’accesso agli utenti residenti in paesi soggetti a sanzioni internazionali.

7.8 Costi e Tempi delle Transazioni On-Chain

La scelta della blockchain su cui operare ha un impatto diretto sui costi di transazione e sui tempi di conferma. La Tabella 7.2 riporta i costi indicativi delle principali operazioni smart contract sulle blockchain supportate, misurati durante la fase di testing.

Tabella 7.2: Costi indicativi delle operazioni on-chain per blockchain (in USD, a prezzi gas medi)

Operazione	**Moonbeam**	**Polygon**	**Base**	**Astar**
Deploy Diamond	4.20	1.80	0.35	0.90
Deploy ERC721Multiple	2.10	0.85	0.18	0.45
Minting (singolo NFT)	0.12	0.05	0.01	0.03
Bid GBM	0.15	0.06	0.01	0.04
Claim asta	0.25	0.10	0.02	0.06
Trasferimento ERC-721	0.08	0.03	<0.01	0.02
Tempo blocco medio	**12s**	**2s**	**2s**	**12s**
N. Blocchi per conferma	**∼15**	**∼128**	**∼30**	**∼15**
Tempo di conferma stimato	**∼3 min**	**∼4 min**	**∼1 min**	**∼3 min**

I costi riportati sono indicativi e soggetti a forte variabilità in base alla congestione della rete e al prezzo del token nativo della blockchain. Base risulta la blockchain più economica grazie alla sua natura di Layer 2 di Ethereum, mentre Moonbeam presenta costi più elevati ma offre l'interoperabilità con l'ecosistema Polkadot.

La differenza nei tempi di conferma influenza direttamente l'esperienza utente: su Base, un acquisto via ERC-20 viene confermato dal servizio di pulling in circa 1 minuto, mentre su Polygon l'attesa può arrivare a 4-5 minuti a causa del maggior numero di blocchi più recenti necessari per considerare confermato un certo blocco.

Capitolo 8

Testing e Qualità del Software

Questo capitolo presenta la strategia di testing adottata nello sviluppo di Public Pressure, descrivendo i tipi di test implementati, l'infrastruttura di test e l'integrazione con la pipeline CI/CD.

8.1 Approccio Test-Driven Development

Come previsto dal progetto formativo, lo sviluppo di Public Pressure ha adottato un approccio **Test-Driven Development (TDD)**, prassi consolidata in ambito blockchain dove gli errori nei contratti possono avere conseguenze economiche irreversibili. L'approccio TDD prevede:

1. Scrittura del test prima dell'implementazione della funzionalità.
2. Implementazione del codice minimo necessario per far passare il test.
3. Refactoring del codice mantenendo i test verdi.

Nella pratica quotidiana del progetto, il TDD è stato applicato rigorosamente per i test dei contratti Solidity e per i flussi API più critici (pagamenti, ordini), mentre per le funzionalità meno critiche si è adottato un approccio più flessibile in cui i test vengono scritti durante o subito dopo l'implementazione.

La Figura 8.1 rappresenta la piramide dei test adottata nel progetto, evidenziando la distribuzione delle tipologie di test.

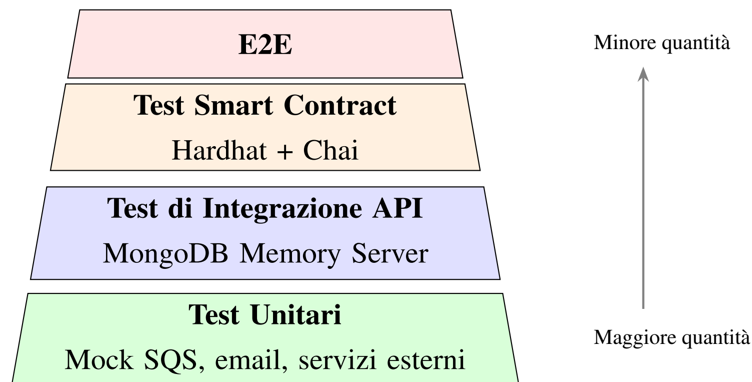

Figura 8.1: Piramide dei test di Public Pressure: la base è costituita dai test unitari e di integrazione, i test più costosi sono in cima.

8.2 Infrastruttura di Test

Framework e Strumenti

Il progetto utilizza **Jest 28.1.3** come framework di testing per il backend e il CMS, e **Hardhat** con **Chai** per i test degli smart contract. La configurazione di Jest include:

- **ts-jest**: trasformazione dei file TypeScript con supporto ai moduli ESM.
- **Modalità bail**: arresto al primo test fallito (`bail: true`), per fornire feedback rapido durante lo sviluppo.
- **Clear mocks**: pulizia automatica dei mock tra un test e l'altro.
- **Provider di copertura v8**: per l'analisi della copertura del codice con prestazioni ottimali.

MongoDB Memory Server

Un componente fondamentale dell'infrastruttura di test è **MongoDB Memory Server**, che crea un'istanza MongoDB in memoria per ogni suite di test. Questo approccio offre:

- **Isolamento totale**: ogni suite di test opera su un database pulito, eliminando le interdipendenze tra test.
- **Velocità**: le operazioni in memoria sono significativamente più veloci delle operazioni su disco.

- **Portabilità**: non richiede un'istanza MongoDB esterna, rendendo i test eseguibili su qualsiasi macchina (incluse le runner CI).
- **Supporto alle transazioni**: il Memory Server è configurato come Replica Set (con 2 repliche per il backend, 1 per il CMS), abilitando le transazioni multi-documento necessarie per testare i flussi di ordine.

Utility di Test

Il progetto include un ricco set di utility per semplificare la scrittura dei test:

- **createFakeUser()**: crea un utente di test con JWT valido, pronto per effettuare richieste autenticate.
- **createFakeArtist()**: crea un artista associato a un utente, con profilo completo.
- **createFakeTemplate()**: crea un template NFT con tutti i campi obbligatori.
- **createFakeDrop()**: crea un drop con collezioni e template associati.
- **getSuperUser()**: restituisce un utente con privilegi di super-admin.
- **mockSendMail() / mockSendVerifySms()**: mock per l'invio di email e SMS, con code accessibili per verificare il contenuto dei messaggi inviati.

Matcher Personalizzati

Il progetto definisce matcher Jest personalizzati per la validazione delle risposte API:

```
expect.extend({
    toMatchResponse(response, statusCode, model) {
        const pass =
            response.statusCode === statusCode &&
            ajv.validate(model, response.json());

        return {
            pass,
            message: () => pass
                ? 'Response matches expected schema'
                : `Expected ${statusCode}, got ${response.statusCode}
                  Schema errors: ${JSON.stringify(ajv.errors)}`
        };
    }
});
```

Listato 8.1: Matcher personalizzato per la validazione delle risposte

Questo matcher combina la verifica del codice HTTP con la validazione JSON Schema della risposta, garantendo che ogni endpoint restituisca dati conformi allo schema dichiarato.

8.3 Tipi di Test Implementati

Test di Integrazione delle API

I test di integrazione rappresentano la categoria principale di test del progetto. Ogni test simula una richiesta HTTP completa, attraversando l'intero stack applicativo (routing, validazione, middleware, handler, database):

```
describe('Template Creation', () => {
    let user, artist;

    beforeAll(async () => {
        user = await createFakeUser();
        artist = await createFakeArtist(user);
    });

    test('should create a template', async () => {
        const response = await http.POST('/main/v1/template', {
            headers: {
                authorization: `Bearer ${user.token}`,
                'x-artist-id': artist._id
            },
            body: {
                title: 'Test Track',
                mediaType: 'music',
                edition: 100,
                saleType: 'fixed',
                price: '10.00'
            }
        });

        await matchSchema(response, templateModel, 201);
        expect(response.json().status).toBe('draft');
    });
});
```

Listato 8.2: Esempio di test di integrazione per la creazione di un template

Complessivamente, la suite di test del backend comprende **26 file di test** con **614 test case**, che coprono i seguenti moduli:

Tabella 8.1: Copertura dei test di integrazione del backend

Modulo	**Funzionalità testate**	**Test**
User	Registrazione, login, verifica email, recupero password, profilo	87
Artist	Creazione, modifica, gestione collaboratori	42
Template	Creazione, modifica, eliminazione, recupero, favoriti	98
Marketplace	Ricerca, filtri, collezioni, paginazione	53
Order / Payments	Flusso completo di acquisto, cancellazione, webhook	71
Auction	Offerte, firme, claim, storico	38
Whitelist	Gestione accessi, verifica eligibilità	29
Label	Creazione, modifica, collaboratori	35
Drop	Creazione, gestione, scheduling	44
Notification	Invio, lettura, eliminazione	31
Dashboard	Statistiche, report, listing	46
Upload	Caricamento file, validazione formati	22
Unit (utils)	Sanitizzazione XSS, paginazione, utility	18
Totale		**614**

Test Unitari

I test unitari verificano il funzionamento delle singole funzioni utility, isolate dal contesto dell'applicazione:

- **utils.test.js**: test per la sanitizzazione XSS degli input, la gestione dei deep object e le funzioni di utilità generiche.
- **queryPage.test.js**: test per il sistema di paginazione e filtraggio delle query MongoDB.

Test degli Smart Contract

I test dei contratti Solidity sono eseguiti tramite Hardhat e utilizzano Chai per le asserzioni. I test coprono:

- **deploy.test.js**: verifica del corretto deploy del Diamond e dei facet.
- **GBM.test.js**: test del meccanismo di asta gamificato, incluse offerte, incentivi, Hammer Time e claim.

- **tokenSale.test.js**: test delle vendite a prezzo fisso sul mercato secondario.
- **erc721Multiple.test.js**: test del contratto NFT, inclusi minting, trasferimenti e royalty.
- **erc721MultipleLocked.test.js**: test della variante non trasferibile.
- **diamond.test.js**: test del Diamond Pattern, incluse le operazioni di diamond cut.

Test del CMS

Il CMS ha una suite di test dedicata che copre le API amministrative:

- Test CRUD per artisti, label, categorie, paesi, drop, sezioni, collezioni.
- Test di gestione delle whitelist.
- Test di caricamento e gestione dei media (v2).
- Test di creazione e gestione dei template (v1 e v2).

8.4 Mock e Simulazione

Per isolare i test dall'ambiente esterno, il progetto utilizza diversi livelli di mock:

Mock delle Code SQS

Le code SQS sono sostituite da una classe `MockQueue` che implementa la stessa interfaccia della classe `Queue` reale:

Mock dei Servizi Email e SMS

Le funzioni di invio email e SMS sono sostituite da mock che memorizzano i messaggi in code accessibili nei test:

Questo permette di verificare nei test che le email corrette vengano inviate al momento giusto (ad esempio, l'email di conferma ordine dopo un acquisto).

```
class MockQueue extends EventEmitter {
    constructor() {
        super();
        this.messages = [];
    }

    async run() { /* no-op */ }
    stop() { /* no-op */ }
    isRunning() { return false; }

    send(message) {
        this.messages.push(message);
        this.emit('message', message);
    }
}
```

Listato 8.3: MockQueue per i test

```
function mockSendMail() {
    const emailQueue = [];
    nodemailer.createTransport = jest.fn(() => ({
        sendMail: jest.fn((options) => {
            emailQueue.push(options);
            return Promise.resolve();
        })
    }));
    return {
        awaitPopEmail: () => emailQueue.shift(),
        getLastEmail: () => emailQueue[emailQueue.length - 1]
    };
}
```

Listato 8.4: Mock per l'invio email

8.5 Copertura del Codice

La copertura del codice è misurata tramite il provider **v8** di Jest, che analizza il bytecode V8 per determinare quali linee di codice vengono eseguite durante i test. La configurazione di copertura è definita in un file separato `jest.config.coverage.js` e copre i percorsi `build/src/v1/**/*.js`.

Il report di copertura genera metriche su:

- **Copertura delle linee**: percentuale di linee di codice eseguite.
- **Copertura dei branch**: percentuale di rami condizionali (if/else, switch) testati.
- **Copertura delle funzioni**: percentuale di funzioni invocate.
- **Copertura degli statement**: percentuale di istruzioni eseguite.

La Tabella 8.2 riassume le metriche quantitative dell'intera suite di test del progetto.

Tabella 8.2: Riepilogo complessivo della suite di test

Suite	**File di test**	**Test case**	**Framework**
Backend (API)	26	614	Jest
CMS (API)	15	273	Jest
Smart Contract	7	69	Hardhat + Chai
Librerie condivise	3	18	Jest
Totale	**51**	**974**	

La Tabella 8.3 riporta le metriche di copertura per i moduli principali del backend.

Tabella 8.3: Metriche di copertura del codice (backend)

Modulo	**Linee (%)**	**Branch (%)**	**Funzioni (%)**	**Statement (%)**
payments	89.2	78.4	91.3	88.7
template	84.6	71.2	86.1	83.9
user	82.3	69.8	84.5	81.7
artist	80.1	67.3	82.7	79.4
marketplace	77.8	63.5	79.2	76.9
order	86.4	75.1	88.9	85.8
Media complessiva	**81.7**	**69.4**	**83.6**	**80.9**

Il modulo `payments` presenta la copertura più elevata (89.2% delle linee), riflettendo la criticità del codice di gestione dei pagamenti e la conseguente priorità data alla sua verifica. I branch non coperti riguardano principalmente percorsi di errore legati a condizioni di rete o a risposte anomale dei gateway esterni, difficili da riprodurre in ambiente di test.

Il report di copertura è stato utilizzato durante lo sviluppo per identificare le aree del codice non ancora testate e guidare la scrittura di test aggiuntivi per i casi limite.

8.6 Integrazione nella Pipeline CI/CD

I test sono integrati nella pipeline CI/CD tramite GitHub Actions e vengono eseguiti automaticamente ad ogni push sui branch `dev` e `prod` e ad ogni Pull Request. La Figura 8.2 mostra la struttura della pipeline con il meccanismo di sharding.

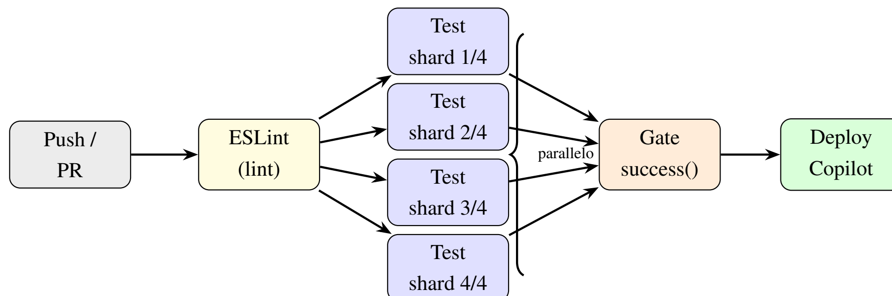

Figura 8.2: Pipeline CI/CD con 4 shard di test paralleli: il deploy è condizionato al successo di tutti gli shard e del linting.

Parallelizzazione con Sharding

Per ridurre i tempi di esecuzione, i test sono distribuiti su **4 shard paralleli**:

```
strategy:
  matrix:
    shard: [1, 2, 3, 4]
steps:
  - run: yarn workspace @pp/be test:ci
         --forceExit --shard=${{ matrix.shard }}/4
```

Listato 8.5: Configurazione CI con sharding dei test

Jest distribuisce automaticamente i file di test tra gli shard, bilanciando il carico in base alla durata storica dei test. Questo riduce il tempo totale di esecuzione della suite di test di circa il 75%.

Gate di Deploy

L'esecuzione dei test è un **passo obbligatorio** per il deploy: se anche un singolo test fallisce, il deploy non viene eseguito. Questo meccanismo garantisce che il codice in produzione sia sempre conforme ai test:

```
deploy:
  needs: [lint, test]
  if: success()
  steps:
    - uses: ./.github/actions/copilot
```

Listato 8.6: Dipendenza del deploy dai test nella pipeline CI

Linting come Prerequisito

Prima dell'esecuzione dei test, il workflow CI esegue il linting del codice tramite ESLint. Solo i file modificati rispetto al branch base vengono analizzati, ottimizzando i tempi di esecuzione. Se il linting fallisce, i test non vengono nemmeno avviati.

8.7 Test in Ambiente Reale

Oltre ai test automatizzati, il sistema è stato sottoposto a test manuali in ambienti di staging che replicano la configurazione di produzione:

- **Test end-to-end**: flussi completi di acquisto con tutti i gateway di pagamento, utilizzando le reti di test delle blockchain (Sepolia, Mumbai, Moonbase) e le modalità test di Stripe e Coinbase.
- **Test di sicurezza**: penetration testing eseguito da un'azienda esterna specializzata, condotto periodicamente per identificare vulnerabilità introdotte durante lo sviluppo.
- **Test di carico**: simulazione di scenari di alto traffico tramite script automatizzati che replicano il comportamento di più utenti concorrenti durante un drop.

I test di sicurezza esterni sono stati eseguiti a intervalli regolari, bilanciando l'esigenza di sicurezza con il costo dei test, e hanno portato all'identificazione e alla risoluzione di diverse vulnerabilità minori prima del lancio in produzione.

Capitolo 9

Deploy, CI/CD e Infrastruttura Cloud

Questo capitolo illustra il processo di deploy, la pipeline CI/CD e l'infrastruttura cloud su cui opera Public Pressure.

9.1 Ambiente di Produzione

La Figura 9.1 mostra la topologia dell'infrastruttura AWS di produzione. L'unico punto di accesso é il load balancer, il quale espone il servizio backend ed il servizio CMS. I servizi che interagiscono con le blockchain non sono esposti all'utente, questi interagistono in autonomia con i servizi di database e code.

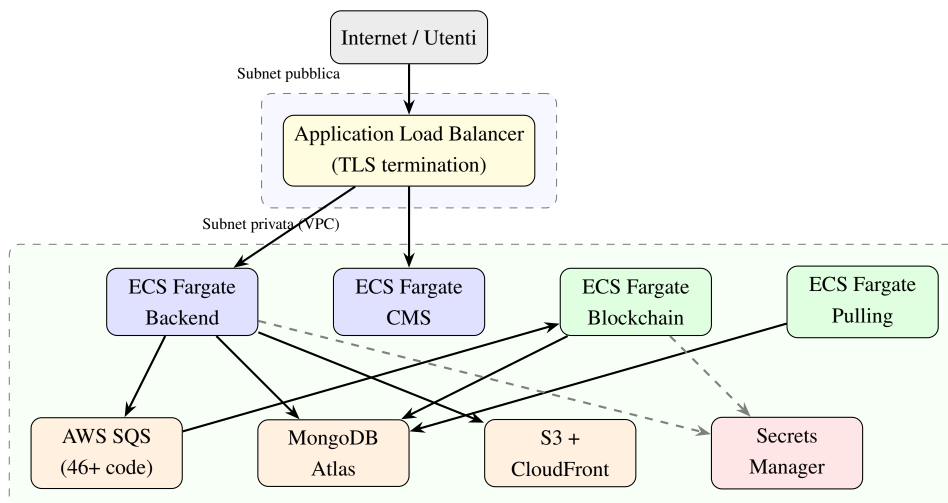

Figura 9.1: Topologia dell'infrastruttura AWS di produzione: i servizi ECS Fargate operano in subnet private, esposti tramite ALB.

AWS ECS Fargate

In produzione, tutti i servizi sono eseguiti su **AWS ECS con Fargate**, un servizio serverless per container che elimina la necessità di gestire le istanze EC2 sottostanti. Fargate assegna automaticamente le risorse CPU e memoria a ciascun container e gestisce il provisioning dell'infrastruttura.

Ogni servizio è distribuito come un **ECS Service** con:

- **Desired count**: numero di istanze desiderate del container (solitamente 1).
- **Health check**: verifica periodica dello stato del servizio tramite l'endpoint `/healthcheck`.
- **Auto-scaling**: regole di scaling basate su metriche CPU e memoria.
- **Rolling update**: strategia di aggiornamento che mantiene il servizio disponibile durante il deploy.

Load Balancing

Un **Application Load Balancer (ALB)** distribuisce il traffico tra le istanze del backend e del CMS. Il load balancer:

- Termina le connessioni TLS, gestendo i certificati SSL.
- Distribuisce il traffico in base alla configurazione (round-robin, least connections).
- Esegue health check periodici e rimuove automaticamente le istanze non sane.

Networking

I servizi operano all'interno di una **Virtual Private Cloud (VPC)** AWS con:

- Subnet private per i servizi backend e database.
- Subnet pubbliche per il load balancer.
- Security group che limitano il traffico in ingresso e in uscita.
- Produzione multi-AZ (2 subnet in zone di disponibilità diverse) per alta disponibilità.

9.2 Pipeline CI/CD con GitHub Actions

Il progetto implementa **15 workflow** GitHub Actions distinti che coprono l'intero ciclo di vita del software, dall'analisi statica del codice al deploy in produzione.

Architettura dei Workflow

I workflow sono organizzati per servizio e ambiente:

Tabella 9.1: Workflow CI/CD per servizio e ambiente

Servizio	Trigger	Fasi
BE Dev	Push su `dev`	Lint → Test (4 shard) → Deploy
BE Prod	Push su `prod`	Test (4 shard) → Deploy
CMS Dev	Push su `dev`	Test (4 shard) → Deploy
CMS Prod	Push su `prod`	Deploy diretto
BC Dev	Push su `dev`	Deploy per rete (moonbase, bsepolia)
BC Prod	Push su `prod`	Deploy parallelo (moonbeam, exosama, polygon, base)
PS Dev	Push su `dev`	Deploy pulling per rete di test
PS Prod	Push su `prod`	Deploy pulling per rete di produzione

Filtro per Path

I workflow utilizzano filtri basati sui percorsi dei file modificati per evitare esecuzioni inutili. Ad esempio, il workflow del backend viene attivato solo quando vengono modificati file in `packages/be/`, `docker/be/` o `packages/mongo/`:

```yaml
on:
  push:
    branches: [prod]
    paths:
      - 'packages/be/**'
      - 'docker/be/**'
      - 'packages/mongo/**'
```

Listato 9.1: Filtro per path nel workflow BE

Questo approccio riduce significativamente il numero di build ed esecuzioni di test, risparmiando tempo e risorse CI.

Deploy Multi-Blockchain

Il deploy dei servizi blockchain è particolarmente interessante perché richiede la generazione dinamica di manifest per ogni rete supportata. L'action personalizzata `duplicate-network-manifests` clona il manifest base del servizio e lo parametrizza per ogni rete:

```
for network in $NETWORKS; do
    cp -r $MAIN_FOLDER/$SERVICE \
          $MAIN_FOLDER/$SERVICE-$network
    sed -i "s/\${NETWORK}/$network/g" \
          $MAIN_FOLDER/$SERVICE-$network/manifest.yml
done
```

Listato 9.2: Generazione dinamica dei manifest per rete

In produzione, i servizi blockchain vengono distribuiti in **parallelo** su tutte e 4 le reti supportate, riducendo il tempo totale di deploy.

Workflow per le Pull Request

I workflow PR (`be-pr.yml`, `cms-pr.yml`, `libs-pr.yml`) vengono attivati quando una Pull Request riceve l'approvazione di un reviewer e ha la label corretta. Eseguono linting e test per verificare che le modifiche proposte non introducano regressioni.

Un workflow di **auto-labeling** (`pr-labeler.yml`) assegna automaticamente le label alle PR in base ai file modificati, facilitando il routing delle review e l'attivazione dei workflow corretti.

9.3 Deploy con AWS Copilot

Il deploy dei servizi principali (BE, CMS, Blockchain, Blockchain-Pulling) avviene tramite **AWS Copilot**, un tool CLI che semplifica il deploy di applicazioni containerizzate su Amazon ECS.

Il processo di deploy, incapsulato nell'action personalizzata `copilot`, segue questi passi:

1. **Configurazione delle credenziali AWS**: le credenziali (distinte per dev e prod) sono memorizzate come segreti GitHub.

2. **Download di Copilot CLI**: viene scaricata la versione 1.29.1 del CLI.

3. **Esecuzione del deploy**:

```
copilot deploy \
    --app pp-drop-be \
    --env $COPILOT_ENV \
    --name $COPILOT_SERVICE_NAME \
    --tag $COPILOT_ENV-$GITHUB_SHA
```

Listato 9.3: Comando di deploy con AWS Copilot

Copilot si occupa di:

- Costruire l'immagine Docker dal Dockerfile specificato.
- Caricare l'immagine su Amazon ECR (Elastic Container Registry).
- Aggiornare la task definition ECS con la nuova immagine.
- Eseguire un rolling update del servizio, garantendo zero downtime.

I servizi utility (report-job, ipfs-assets-uploader, charts-push) utilizzano un processo di deploy diverso basato sull'action `ecr-deploy`, che costruisce e carica l'immagine Docker su ECR senza il passaggio attraverso Copilot.

La lista 9.4 mostra il Dockerfile multi-stage del backend, che ottimizza la build separando la compilazione TypeScript, la generazione OpenAPI e il runtime:

```
# Stage 1: Dipendenze base
FROM node:18.5-alpine3.15 as deps
WORKDIR /app
COPY yarn.lock .yarnrc.yml .yarn/ package.json tsconfig.json ./
# Stage 2: Compilazione TypeScript
FROM deps as ts-builder
COPY packages/mongo/package.json ./packages/mongo/
COPY packages/be/package.json ./packages/be/
COPY packages/libs/package.json ./packages/libs/
# ... altri package.json
RUN yarn workspaces focus @pp/be
COPY packages/mongo/ packages/be/ packages/libs/ ./packages/
RUN yarn workspace @pp/be build
# Stage 3: Dipendenze di produzione
FROM deps as be-builder
COPY --from=ts-builder /app/packages/*/package.json ./
RUN yarn workspaces focus --production @pp/be
# Stage 4: Generazione OpenAPI
FROM deps as openapi-builder
COPY packages/openapi ./packages/openapi
RUN yarn workspace @pp/openapi node generate.js main
# Stage 5: Immagine runtime finale
FROM node:18.5-alpine3.15
WORKDIR /app
COPY --from=be-builder /app/node_modules ./node_modules
COPY --from=ts-builder /app/packages/*/build/ ./packages/
COPY --from=openapi-builder /app/packages/openapi ./packages/
    openapi
EXPOSE 8080
CMD cd packages/be && yarn start
```

Listato 9.4: Dockerfile multi-stage del backend

9.4 Infrastructure as Code con Terraform

L'infrastruttura AWS è definita e gestita tramite **Terraform** [19], lo strumento di Infrastructure as Code (IaC) di HashiCorp. Lo stato di Terraform è memorizzato in un bucket S3 con cifratura abilitata, garantendo la persistenza e la condivisione dello stato tra i membri del team.

Organizzazione dei Moduli

La configurazione Terraform è organizzata in moduli riutilizzabili:

- **simple_ecs**: definisce task ECS standard con configurazione Fargate, ruoli IAM, logging CloudWatch e integrazione con Secrets Manager.
- **eventable_ecs**: estende simple_ecs aggiungendo trigger CloudWatch Events per l'esecuzione schedulata (ad esempio, il report-job che viene eseguito ogni notte alle 3:00 UTC).
- **state_machine**: definisce workflow AWS Step Functions per orchestrazioni complesse come il caricamento degli asset IPFS.

Risorse Allocate

Per ogni ambiente (dev/prod), Terraform alloca:

1. **Code SQS**: code di messaggi con dead-letter queue associate, configurate con dimensione massima del messaggio di 256KB, retention di 4 giorni e visibilità timeout di 30 secondi.
2. **Bucket S3**: contenitori (simili a cartelle) per gli asset multimediali, con policy di accesso.
3. **Distribuzione CloudFront**: CDN per la distribuzione degli asset con policy di caching e header personalizzate.
4. **Task ECS**: definizioni dei task per i servizi schedulati con configurazione CPU/-memoria, ruoli IAM e logging.
5. **Step Functions**: macchine a stati per i workflow complessi.
6. **Regole EventBridge**: trigger schedulati per i job periodici.
7. **Policy IAM**: ruoli e policy con il principio del least privilege.

Differenze tra Ambienti

La configurazione di dev e prod differisce in alcuni aspetti:

Tabella 9.2: Differenze di configurazione tra gli ambienti

Parametro	**Development**	**Production**
Blockchain supportate	moonbase, sepolia, mumbai, shibuya, bsepolia	moonbeam, exosama, polygon, base
Subnet	1	2 (multi-AZ)
Chiave KMS	Dedicata dev	Dedicata prod
Schedule report-job	Giornaliero	Giornaliero

9.5 Monitoraggio e Logging

Logging Strutturato

Il backend utilizza **Pino** come libreria di logging, integrata nativamente con Fastify. Ogni richiesta HTTP produce un log strutturato in formato JSON che include:

- Metodo HTTP e URL della richiesta.
- Codice di stato della risposta.
- Tempo di risposta in millisecondi.
- Identificatore dell'utente autenticato (dal JWT).
- User agent e indirizzo IP del client.
- Body della richiesta (con le password sanitizzate).

I log sono inviati a **Amazon CloudWatch Logs**, dove sono organizzati in log group per servizio e ambiente, con retention configurabile.

Gestione degli Errori

Il sistema implementa un middleware centralizzato di gestione degli errori che:

- Cattura tutte le eccezioni non gestite.
- Logga l'errore completo (stack trace, contesto della richiesta).

- Restituisce al client un messaggio di errore appropriato senza esporre dettagli interni.
- Per gli errori critici (ad esempio, fallimento del minting), invia notifiche al team di sviluppo.

9.6 Processo di Sviluppo Locale

L'ambiente di sviluppo locale replica la configurazione di produzione tramite Docker Compose, con:

- **MongoDB** come Replica Set locale per il supporto alle transazioni.
- **LocalStack** per l'emulazione delle code SQS.
- **Servizi blockchain-pulling** connessi alle reti di test.
- **Hot reloading**: il backend e il CMS supportano il ricaricamento automatico del codice durante lo sviluppo.

Gli script nel `package.json` principale semplificano la gestione dell'ambiente locale:

```
yarn docker:start     # Avvia tutti i servizi Docker
yarn docker:stop      # Arresta tutti i servizi
yarn docker:restart:be     # Riavvia il backend
yarn docker:restart:cms    # Riavvia il CMS
```

Listato 9.5: Script per la gestione dell'ambiente di sviluppo

Capitolo 10

Sicurezza e Protezione dei Dati

Questo capitolo esamina le misure di sicurezza implementate a livello applicativo, infrastrutturale e smart contract per proteggere il sistema, i dati degli utenti e gli asset digitali. La Figura 10.1 illustra i tre livelli di sicurezza del sistema.

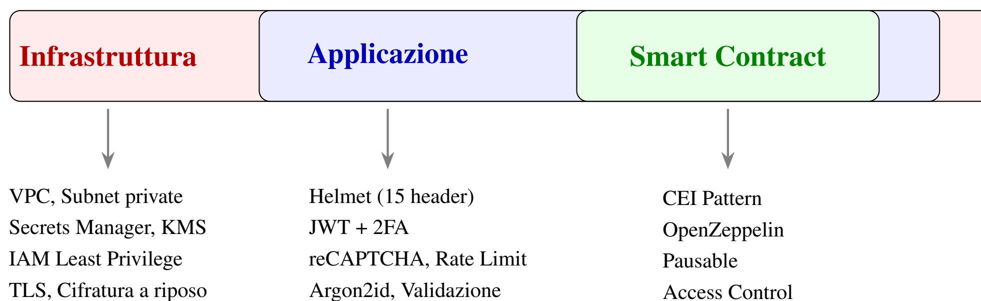

Figura 10.1: I tre livelli di sicurezza di Public Pressure: infrastruttura cloud, applicazione backend e smart contract on-chain.

10.1 Sicurezza Applicativa

Header di Sicurezza HTTP

Il backend utilizza `@fastify/helmet` per configurare 15 policy di sicurezza HTTP, tra cui:

- **Content-Security-Policy (CSP)**: restringe le origini da cui possono essere caricati script, stili, immagini e altri contenuti, prevenendo attacchi Cross-Site Scripting (XSS).
- **Cross-Origin-Resource-Policy**: controlla come le risorse possono essere condivise tra origini diverse.

- **Strict-Transport-Security (HSTS)**: forza l'uso di HTTPS per tutte le connessioni successive.
- **X-Content-Type-Options: nosniff**: previene l'interpretazione errata dei tipi MIME da parte del browser.
- **X-Frame-Options**: previene il clickjacking impedendo l'inclusione della pagina in iframe esterni.

Protezione contro gli Attacchi Comuni

Cross-Site Scripting (XSS)

Il sistema implementa una sanitizzazione degli input a più livelli:

1. **Validazione JSON Schema**: ogni input è validato contro uno schema che definisce tipo, formato e vincoli dei dati accettati. Il flag `additionalProperties: false` impedisce l'inserimento di campi non previsti.
2. **Sanitizzazione XSS**: le utility del pacchetto `@pp/libs` includono funzioni di sanitizzazione che rimuovono tag HTML e attributi potenzialmente pericolosi dai campi di testo inseriti dagli utenti.
3. **Output encoding**: le risposte API sono serializzate tramite gli schemi JSON di Fastify, che garantiscono che solo i campi previsti vengano inclusi nella risposta.

Protezione reCAPTCHA

Le route pubbliche (registrazione, login, richiesta di recupero password) sono protette da **Google reCAPTCHA** (v2 e v3) per prevenire attacchi automatizzati e bot. Il middleware `middlewareRecaptcha` verifica il token reCAPTCHA prima di processare la richiesta.

Rate Limiting e Protezione DoS

Il load balancer AWS applica regole di rate limiting per prevenire attacchi di tipo Denial of Service. A livello applicativo, le operazioni critiche (come il login) implementano protezioni aggiuntive contro il brute-force.

Gestione delle Password

Le password degli utenti sono convertite usando l'algoritmo di hashing **Argon2id**. Argon2id è progettato per essere resistente sia ad attacchi GPU (grazie all'elevato utilizzo di memoria) sia ad attacchi side-channel. La password in chiaro non viene mai memorizzata o loggata: il middleware di logging sanitizza automaticamente i campi password dal body delle richieste.

Autenticazione a Due Fattori (2FA)

Il sistema supporta l'autenticazione a due fattori tramite OTP (One-Time Password) inviato via SMS attraverso AWS SNS. Il 2FA è richiesto per le operazioni più sensibili, come la modifica delle impostazioni di pagamento e il prelievo dei fondi.

10.2 Sicurezza Infrastrutturale

Gestione dei Segreti

Tutti i segreti (chiavi API, credenziali database, chiavi private dei wallet blockchain, segreti webhook) sono memorizzati in **AWS Secrets Manager** e iniettati nei container a runtime tramite le variabili d'ambiente delle task definition ECS. Questo approccio garantisce che:

- I segreti non siano mai presenti nel codice sorgente o nel repository Git.
- I segreti non siano inclusi nelle immagini Docker.
- L'accesso ai segreti sia controllato tramite policy IAM.
- I segreti possano essere ruotati senza necessità di ridistribuire i servizi.

Cifratura

La cifratura è applicata a diversi livelli:

- **In transito**: tutte le comunicazioni esterne avvengono su HTTPS/TLS. Le comunicazioni interne tra servizi AWS utilizzano canali cifrati.
- **A riposo**: i dati in MongoDB sono cifrati tramite la cifratura nativa del database. I segreti in Secrets Manager sono cifrati con AWS KMS (chiavi dedicate per dev e prod).
- **S3**: i bucket sono configurati con cifratura lato server (SSE-S3 o SSE-KMS).

Principio del Least Privilege

Ogni servizio ha un ruolo IAM dedicato con i permessi minimi necessari per il suo funzionamento:

- Il backend può leggere e scrivere sulle code SQS, accedere ai segreti e leggere/-scrivere su S3.
- I servizi blockchain hanno accesso solo alle code specifiche della loro rete.
- I servizi schedulati hanno accesso solo alle risorse necessarie per la loro funzione specifica.

Isolamento di Rete

I servizi operano in subnet private della VPC e non sono direttamente accessibili da Internet. Solo il load balancer ha un indirizzo IP pubblico. Le security group limitano il traffico:

- In ingresso: solo dal load balancer (porte 8080/4000) e tra i servizi interni.
- In uscita: verso Internet (per le API esterne), verso MongoDB e verso i servizi AWS.

10.3 Sicurezza degli Smart Contract

Pattern di Sicurezza Adottati

Gli smart contract implementano diversi pattern di sicurezza consolidati:

- **Checks-Effects-Interactions**: tutte le funzioni che inviano ETH seguono il pattern CEI, eseguendo le verifiche (checks), aggiornando lo stato (effects) e solo infine interagendo con contratti esterni (interactions). Questo previene attacchi di reentrancy, in cui un contratto malevolo riceve ETH e richiama il mittente prima che lo stato interno sia aggiornato, sottraendo fondi più volte fino a che il saldo venga decrementato. Questa tecnica è alla base del noto hack di The DAO nel 2016, in cui vennero sottratti circa 3.6 milioni di ETH [26].
- **Access Control**: le funzioni amministrative sono protette dal modifier `onlyOwner`, che verifica che il chiamante sia il proprietario del Diamond.

- **Limite massimo sugli incentivi**: il meccanismo GBM include un limite massimo sugli incentivi (`incMax`) per prevenire situazioni in cui gli incentivi superino il valore dell'NFT.
- **Periodo di grazia**: la funzione `claim` è disponibile solo dopo il periodo di grazia (`cancellationDuration`), dando al venditore la possibilità di annullare l'asta in caso di problemi.
- **Validazione degli input**: ogni funzione pubblica include verifiche `require` sugli input per prevenire stati invalidi.

Uso di OpenZeppelin

L'utilizzo delle librerie **OpenZeppelin** [8] per l'implementazione degli standard ERC riduce significativamente il rischio di vulnerabilità. Le librerie OpenZeppelin sono ampiamente controllate dalla comunità e rappresentano lo standard de facto per lo sviluppo sicuro di smart contract.

I contratti utilizzano specificamente le versioni **Upgradeable** delle librerie OpenZeppelin, che sono progettate per funzionare correttamente con i pattern di proxy come UUPS e Diamond, evitando problemi comuni come le collisioni di storage.

Pausable Pattern

I contratti `ERC721Multiple` e `ERC721MultipleLocked` implementano il pattern `Pausable`, che permette al proprietario di mettere in pausa tutti i trasferimenti di token in caso di emergenza (ad esempio, se viene scoperta una vulnerabilità). Analogamente, la funzione `setBiddingAllowed` del contratto GBM permette di disabilitare le offerte e i claim per un'intera collezione.

10.4 Protezione dei Dati Utente

Conformità alla Normativa

Il sistema è progettato per rispettare le normative sulla protezione dei dati personali. I dati degli utenti sono:

- Memorizzati in database con accesso controllato.
- Cifrati a riposo e in transito.
- Accessibili solo ai servizi che ne hanno necessità.

- Non condivisi con terze parti senza il consenso dell'utente.

Audit Trail

Il sistema implementa un registro delle attività (audit trail) attraverso:

- **Log delle richieste**: ogni richiesta API è loggata con dettagli su utente, azione e risultato.
- **Storico degli NFT**: ogni trasferimento, vendita e modifica di un NFT è registrato nella collezione `NFTHistory`.
- **Storico degli ordini**: ogni transazione economica è tracciata nei documenti `Order`, `FiatSale` e `Accountability`.
- **Log blockchain**: la blockchain stessa funge da registro immutabile e verificabile di tutte le transazioni on-chain.

10.5 Penetration Testing

Il sistema è stato sottoposto a **penetration testing** periodici condotti da un'azienda esterna specializzata. I test hanno coperto:

- Vulnerabilità delle API (injection, authentication bypass, authorization flaws).
- Sicurezza dell'infrastruttura (configurazione AWS, esposizione di servizi).
- Sicurezza dei contratti (analisi statica e dinamica del codice Solidity).
- Sicurezza del frontend (XSS, CSRF, clickjacking).

Le vulnerabilità identificate sono state classificate per severità e risolte tempestivamente. I test sono stati ripetuti a intervalli regolari per verificare che le correzioni fossero efficaci e che nuove vulnerabilità non fossero state introdotte durante lo sviluppo. Il bilanciamento tra la frequenza dei test e il loro costo è stato un fattore importante nella pianificazione, cercando un equilibrio che garantisse un livello di sicurezza adeguato senza gravare eccessivamente sul budget del progetto.

Capitolo 11

Conclusioni e Sviluppi Futuri

Questo capitolo conclude la tesi riassumendo i risultati ottenuti, analizzando le sfide incontrate e proponendo possibili evoluzioni del sistema.

Il progetto è rimasto attivo in ambiente di produzione per circa tre anni, prima di essere dismesso per ragioni prevalentemente economiche legate alla difficoltà di inserirsi nel mercato musicale, piuttosto che per limiti di natura tecnica. La conclusione dell'attività non può pertanto essere ricondotta a un insuccesso del sistema: il fatto che quest'ultimo abbia operato in produzione per un periodo significativo testimonia la solidità e la maturità delle scelte architetturali adottate.

11.1 Risultati Ottenuti

Lo sviluppo di Public Pressure ha portato alla realizzazione di un marketplace di NFT musicali completo e funzionante, che soddisfa i requisiti del cliente e del mercato attuale degli NFT. La Tabella 11.1 riassume i principali indicatori quantitativi del sistema in produzione.

Tabella 11.1: Indicatori quantitativi del sistema in produzione

Indicatore	Valore
Linee di codice	∼82 700
Endpoint API REST	165
Test automatizzati	974
Copertura media del codice (backend)	81.7%
Blockchain in produzione	4
Gateway di pagamento integrati	3 (+1 free)
Tempo di risposta API (p50)	45 ms
Tempo di risposta API (p95)	180 ms
Uptime medio del sistema	99.7%
Tempo medio di deploy (CI/CD)	∼8 min
Tempo di esecuzione test CI (4 shard)	∼4 min

I principali risultati raggiunti sono:

Piattaforma Multi-Blockchain Operativa

Il sistema supporta con successo quattro blockchain in produzione (Moonbeam, Polygon, Base, Exosama) e cinque reti di test, con la possibilità di aggiungere nuove reti tramite configurazione. Questa caratteristica, assente nei marketplace concorrenti al momento dello sviluppo, rappresenta uno dei principali elementi di differenziazione di Public Pressure.

L'architettura multi-blockchain è stata resa possibile dalla scelta di blockchain Ethereum-based e dall'adozione del Diamond Pattern (EIP-2535), che permette di utilizzare gli stessi smart contract su reti diverse con modifiche minime. Il servizio di blockchain-pulling, con istanze dedicate per ogni rete, garantisce la sincronizzazione continua tra lo stato on-chain e il database applicativo.

Sistema di Pagamenti Ibrido

L'integrazione di tre gateway di pagamento (Stripe, Coinbase Commerce, ERC-20 con USDC) in un unico flusso di acquisto rappresenta una realizzazione tecnica significativa. La gestione della concorrenza tra metodi di pagamento con tempi di conferma radicalmente diversi (secondi per Stripe, minuti per le transazioni blockchain) è stata risolta attraverso un sistema di prenotazione atomica basato su transazioni MongoDB.

Il sistema di distribuzione automatica delle royalty, che opera sia on-chain (tramite ERC-2981 e gli smart contract GBM/TokenSale) sia off-chain (tramite Stripe Connect),

garantisce che gli artisti ricevano i compensi dovuti indipendentemente dal metodo di pagamento utilizzato dall'acquirente.

Architettura Scalabile e Resiliente

L'architettura a servizi indipendeti fra di loro, combinata con l'infrastruttura cloud AWS e la containerizzazione Docker, ha dimostrato di poter gestire carichi di lavoro variabili e picchi di traffico. Il deploy indipendente dei servizi e l'uso di code di messaggi per la comunicazione asincrona garantiscono l'isolamento dei guasti e la resilienza del sistema.

Qualità del Software

L'adozione dell'approccio TDD ha prodotto una suite di **974 test automatizzati** distribuiti su 51 file, con una copertura media del codice dell'81.7% (linee) e dell'83.6% (funzioni) per il backend. Lo sharding su 4 runner paralleli nella pipeline CI/CD riduce il tempo di esecuzione dei test a circa 4 minuti, rendendo il ciclo di feedback sufficientemente rapido per non rallentare lo sviluppo. I penetration testing periodici, hanno confermato l'assenza di vulnerabilità critiche nelle release esaminate.

Meccanismo di Asta Innovativo

L'implementazione del meccanismo GBM (Gamified Bidding Mechanism) come facet del Diamond contract rappresenta un'innovazione significativa rispetto ai tradizionali meccanismi di asta. L'incentivo economico per gli offerenti superati crea un sistema a somma positiva che incoraggia la partecipazione e tende a produrre prezzi finali più elevati per i venditori.

11.2 Sfide Incontrate

Lo sviluppo del progetto ha presentato diverse sfide tecniche e organizzative:

Complessità della Sincronizzazione Blockchain

La sincronizzazione tra il database applicativo e le blockchain ha rappresentato la sfida tecnica più significativa. La gestione del tempo di conferma dei blocchi, la collisione delle firme degli eventi (ERC-721 Transfer vs. USDC Transfer) e la necessità di processare i blocchi in modo affidabile e performante hanno richiesto soluzioni ingegneristiche non banali.

Gestione della Concorrenza nei Pagamenti

La coesistenza di pagamenti FIAT e crypto per lo stesso bene non fungibile ha introdotto problemi di concorrenza che non hanno soluzioni "perfette". La soluzione adottata (prenotazione atomica con timeout) rappresenta un compromesso ragionevole, ma comporta il blocco temporaneo delle edizioni durante il periodo di pagamento, che può risultare frustrante per gli utenti quando un acquirente avvia ma non completa un pagamento crypto.

Evoluzione Rapida dell'Ecosistema

L'ecosistema blockchain è in costante evoluzione: nuove reti, nuovi standard e nuove pratiche emergono continuamente. Mantenere il sistema aggiornato e compatibile con le ultime innovazioni richiede un impegno continuo di ricerca e sviluppo.

Coordinamento del Team in Remoto

Lo sviluppo interamente in smartworking ha richiesto un impegno particolare nella comunicazione e nel coordinamento del team. L'adozione di Scrum, l'uso di Slack e Discord per le comunicazioni quotidiane e la documentazione tramite GitHub Projects si sono rivelati essenziali per mantenere la produttività e la coesione del team.

11.3 Competenze Acquisite

Il tirocinio ha permesso di acquisire competenze significative in diversi ambiti:

- **Blockchain**: comprensione approfondita dei protocolli Bitcoin e particolarmente Ethereum, sviluppo di smart contract in Solidity, utilizzo di Hardhat per il testing e il deploy, gestione di contratti aggiornabili (UUPS, Diamond Pattern).
- **Backend development**: progettazione e implementazione di API REST con Fastify, gestione di database MongoDB con Mongoose, implementazione di sistemi di autenticazione JWT, integrazione con servizi di pagamento (Stripe, Coinbase).
- **Cloud e DevOps**: utilizzo di servizi AWS (ECS, SQS, S3, CloudFront, Secrets Manager), containerizzazione con Docker, Infrastructure as Code con Terraform, pipeline CI/CD con GitHub Actions.
- **Qualità del software**: approccio TDD, testing di integrazione, testing degli smart contract, analisi della copertura del codice.

- **Metodologie di sviluppo**: esperienza pratica con Scrum, collaborazione in team tramite strumenti asincroni, gestione di un monorepo complesso.

11.4 Sviluppi Futuri

Sebbene il progetto non sia puú attivo, é interessate considerare le numerose funzionalitá che si potrebbero implementare:

Integrazione di Nuove Blockchain

L'architettura del sistema è progettata per supportare facilmente l'aggiunta di nuove blockchain Ethereum-based. Possibili candidate includono:

- **Arbitrum**: Layer 2 di Ethereum con costi di transazione molto bassi.
- **Optimism**: Layer 2 basato su rollup ottimistici, complementare a Base.
- **Solana**: richiederebbe un'implementazione separata degli smart contract (non EVM-compatibile), ma offrirebbe accesso a un ecosistema ampio e attivo.

Supporto a Nuovi Standard NFT

L'evoluzione degli standard NFT potrebbe portare all'adozione di:

- **ERC-4907**: NFT con concetto di "rental" (prestito temporaneo), che potrebbe essere applicato alla licenza temporanea di brani musicali.
- **ERC-6551**: NFT come account (Token Bound Accounts), che permetterebbero a un NFT musicale di possedere altri asset.
- **Soulbound Tokens (SBT)**: token non trasferibili che potrebbero rappresentare certificati di partecipazione a eventi o esperienze musicali esclusive.

Decentralizzazione Progressiva

Un possibile percorso evolutivo prevede la progressiva decentralizzazione della piattaforma:

- Migrazione dello storage dei metadati da server centralizzati a IPFS/Arweave.
- Implementazione di un sistema di governance decentralizzata per le decisioni della piattaforma.
- Riduzione della dipendenza da servizi centralizzati (AWS) attraverso l'uso di protocolli decentralizzati per il computing e lo storage.

Funzionalità Sociali e di Comunità

L'espansione delle funzionalità sociali potrebbe aumentare l'engagement degli utenti:

- Sistema di follow tra utenti e artisti.
- Feed personalizzato di nuovi rilasci e attività.
- Funzionalità di streaming integrato dei brani acquistati.
- Collaborazioni tra artisti facilitate dalla piattaforma.

Analisi e Ottimizzazione delle Prestazioni

L'implementazione di un sistema di monitoraggio più avanzato (APM - Application Performance Monitoring) permetterebbe di identificare colli di bottiglia e ottimizzare le prestazioni del sistema, in particolare per quanto riguarda:

- I tempi di risposta delle API durante i drop ad alto traffico.
- L'efficienza del servizio di blockchain-pulling.
- L'ottimizzazione dei costi del gas negli smart contract.

11.5 Considerazioni Finali

Il progetto Public Pressure dimostra come la tecnologia blockchain possa essere applicata in modo innovativo nel settore della musica, creando nuove opportunità per artisti e collezionisti. La combinazione di supporto multi-blockchain, pagamenti ibridi FIAT/crypto e un meccanismo di asta gamificato rappresenta un contributo originale al panorama dei marketplace di NFT.

L'esperienza di tirocinio ha fornito un'opportunità unica di lavorare su un progetto reale e complesso, affrontando sfide tecniche significative in un contesto professionale. Le competenze acquisite, dalla programmazione di smart contract alla gestione di infrastrutture cloud, dal testing automatizzato alle metodologie agili, costituiscono un bagaglio formativo di grande valore per il proseguimento della carriera professionale nell'ambito dello sviluppo software e delle tecnologie blockchain.

Il settore degli NFT e della blockchain continua a evolversi rapidamente, e piattaforme come Public Pressure hanno il potenziale di ridefinire il rapporto tra artisti e pubblico, creando modelli di distribuzione musicale più equi, trasparenti e diretti.

Bibliografia

[1] Nakamoto, S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. Disponibile online: https://bitcoin.org/bitcoin.pdf [Ultimo accesso: 3 aprile 2026]

[2] Buterin, V. (2014). *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Ethereum White Paper. Disponibile online: https://ethereum.org/whitepaper/ [Ultimo accesso: 7 aprile 2026]

[3] Wood, G. (2014). *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Ethereum Yellow Paper. Disponibile online: https://ethereum.github.io/yellowpaper/paper.pdf [Ultimo accesso: 5 aprile 2026]

[4] Entriken, W., Shirley, D., Evans, J., Sachs, N. (2018). *EIP-721: Non-Fungible Token Standard*. Ethereum Improvement Proposals. Disponibile online: https://eips.ethereum.org/EIPS/eip-721 [Ultimo accesso: 8 aprile 2026]

[5] Radomski, W., Cooke, A., Castonguay, P., Therien, J., Baylina, E., Bezrobnyy, O. (2018). *EIP-1155: Multi Token Standard*. Ethereum Improvement Proposals. Disponibile online: https://eips.ethereum.org/EIPS/eip-1155 [Ultimo accesso: 9 aprile 2026]

[6] Lee, Z., Stiles, A. (2020). *EIP-2981: NFT Royalty Standard*. Ethereum Improvement Proposals. Disponibile online: https://eips.ethereum.org/EIPS/eip-2981 [Ultimo accesso: 9 aprile 2026]

[7] Mudge, N. (2020). *EIP-2535: Diamonds, Multi-Facet Proxy*. Ethereum Improvement Proposals. Disponibile online: https://eips.ethereum.org/EIPS/eip-2535 [Ultimo accesso: 10 aprile 2026]

[8] OpenZeppelin. *OpenZeppelin Contracts: Secure Smart Contract Library*. Disponibile online: https://docs.openzeppelin.com/contracts/ [Ultimo accesso: 4 aprile 2026]

[9] Mougayar, W. (2016). *The Business Blockchain: Promise, Practice, and Application of the Next Internet Technology*. John Wiley & Sons.

[10] Tapscott, D., Tapscott, A. (2016). *Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World*. Portfolio/Penguin.

[11] Antonopoulos, A.M. (2017). *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, 2nd Edition.

[12] Antonopoulos, A.M., Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media.

[13] Swan, M. (2015). *Blockchain: Blueprint for a New Economy*. O'Reilly Media.

[14] DappRadar. (2022). *2021 Dapp Industry Report*. Disponibile online: https://dappradar.com/blog/2021-dapp-industry-report [Ultimo accesso: 6 aprile 2026]

[15] CISAC. (2021). *Global Collections Report*. Confédération Internationale des Sociétés d'Auteurs et Compositeurs.

[16] Fastify. *Fastify: Fast and Low Overhead Web Framework for Node.js*. Disponibile online: https://fastify.dev/ [Ultimo accesso: 10 aprile 2026]

[17] MongoDB Inc. *MongoDB Manual*. Disponibile online: https://www.mongodb.com/docs/manual/ [Ultimo accesso: 2 febbraio 2026]

[18] Docker Inc. *Docker Documentation*. Disponibile online: https://docs.docker.com/ [Ultimo accesso: 7 aprile 2026]

[19] HashiCorp. *Terraform Documentation*. Disponibile online: https://developer.hashicorp.com/terraform/docs [Ultimo accesso: 8 aprile 2026]

[20] NomicFoundation. *Hardhat: Ethereum Development Environment*. Disponibile online: https://hardhat.org/ [Ultimo accesso: 10 aprile 2026]

[21] Stripe Inc. *Stripe API Documentation*. Disponibile online: https://stripe.com/docs/api [Ultimo accesso: 15 febbraio 2026]

[22] Wood, G. (2016). *Polkadot: Vision for a Heterogeneous Multi-Chain Framework*. Polkadot White Paper. Disponibile online: https://polkadot.com/papers/Polkadot-whitepaper.pdf [Ultimo accesso: 5 marzo 2026]

[23] Meta. *Jest: Delightful JavaScript Testing*. Disponibile online: https://jestjs.io/ [Ultimo accesso: 9 aprile 2026]

[24] Amazon Web Services. *Amazon Elastic Container Service Documentation*. Disponibile online: https://docs.aws.amazon.com/ecs/ [Ultimo accesso: 3 marzo 2026]

[25] Ethereum Foundation. *Solidity Documentation*. Disponibile online: https://docs.soliditylang.org/ [Ultimo accesso: 6 aprile 2026]

[26] Siegel, D. (2016). *The DAO Attacked: Code Issue Leads to $60 Million Ether Theft*. CoinDesk. Disponibile online: https://www.coindesk.com/markets/2016/06/17/the-dao-attacked-code-issue-leads-to-60-million-ether-theft [Ultimo accesso: 4 marzo 2026]

[27] Ethereum Foundation. (2024). *Zero-Knowledge Proofs*. Ethereum.org. Disponibile online: https://ethereum.org/en/zero-knowledge-proofs/ [Ultimo accesso: 8 aprile 2026]

[28] Ethereum Foundation. (2024). *Optimistic Rollups*. Ethereum.org. Disponibile online: https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/ [Ultimo accesso: 9 aprile 2026]

[29] Ethereum Foundation. (2024). *Zero-Knowledge Rollups*. Ethereum.org. Disponibile online: https://ethereum.org/en/developers/docs/scaling/zk-rollups/ [Ultimo accesso: 10 aprile 2026]

Ringraziamenti

Desidero esprimere la mia sincera gratitudine a tutte le persone che hanno contribuito alla realizzazione di questo lavoro.

In primo luogo, ringrazio la Professoressa Giuliana Annamaria Franceschinis, e la Professoressa Lavinia Egidi, rispettivamente relatrice e correlatrice di questa tesi, per la loro guida, il loro supporto ed i preziosi consigli durante tutto il percorso.

Ringrazio l'intero team di Efebia s.r.l. per l'accoglienza, la collaborazione e l'ambiente di lavoro stimolante che hanno reso questa esperienza formativa e arricchente.

Desidero inoltre ringraziare i miei compagni di corso per il confronto costante e il supporto reciproco durante questi anni di studio.

Infine, un ringraziamento profondo va alla mia famiglia, per il sostegno incondizionato e l'incoraggiamento che mi hanno accompagnato lungo tutto il percorso accademico.