ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea in Ingegneria Informatica

La Blockchain e la sua integrazione nei database relazionali per il trattamento dei dati sensibili

Relatore:
Chiar.mo Prof.
Paolo Ciaccia

Presentata da:
Francesco Galli

Sessione dicembre 2025
Anno Accademico 2024/2025

All'ombra che da sempre mi guarda,
e che mi ha sempre detto
che non ce l'avrei mai fatta.

Sommario

Il contesto della sicurezza e tracciabilità dei dati rappresenta uno degli aspetti più importanti dell'era della digitalizzazione. Sono molteplici gli ambiti in cui l'integrazione di database richiede delle garanzie di sicurezza. Alcuni esempi sono sicuramente l'ambito sanitario, giuridico o finanziario. Questa tesi si concentra sullo studio e la realizzazione di un database relazionale per il trattamento dei dati sensibili, che sfrutti e integri la tecnologia Blockchain. L'obiettivo principale è lo sviluppo di un modello basato sulla struttura e sull'efficienza tipica dei database relazionali, che al contempo garantisca l'immutabilità e trasparenza delle operazioni effettuate sui dati (inserimento, modifica o cancellazione) attraverso un modello di realizzazione tipico delle blockchain. Nei seguenti capitoli verrà riportato un approfondimento introduttivo sui database relazionali, sulla tecnologia Blockchain e l'integrazione di questa in un modello esemplificativo.

Indice

Listings

Elenco delle figure

Capitolo 1

Introduzione

La contemporanea era digitale è caratterizzata da una crescita esponenziale dei dati da salvaguardare, la cui gestione, integrità e sicurezza rappresentano delle sfide architetturali di primaria importanza. Queste problematiche hanno messo in luce i limiti delle ormai diffusissime tecnologie di archiviazione centralizzata.

La centralizzazione dei dati, ovvero il processo di raccolta e archiviazione dei dati provenienti da fonti diverse locate in un unico sistema (repository) centrale, porta con sé dei limiti non indifferenti sotto vari punti di vista. Dal guasto su un server, al collo di bottiglia delle prestazioni, alla scalabilità limitata, ai rischi di sicurezza e privacy, ai costi elevati, si evince che sono molteplici le criticità apportate da questa tipologia di architettura.

Da decenni il modello relazionale risulta essere uno dei fondamenti dell'archiviazione strutturata, implementato tramite Database Management System (DBMS) centralizzati. Nonostante ad oggi sia il metodo di archiviazione più diffuso negli ultimi decenni, e offra indiscutibili vantaggi in termini di efficienza interrogativa e maturità tecnologica, è molto importante prendere in considerazione le vulnerabilità intrinseche legate alla sua natura centralizzata. L'intera fiducia del sistema è fondata su un'unica autorità amministrativa, che può costituire un *single point of failure*. Quest'ultimo può essere preso di mira in attacchi informatici, manipolazioni interne o guasti hardware. L'avvento delle nuove tecnologie ha portato delle alternative efficaci alle limitazioni apportate dalla centralizzazione, mantenendo i vantaggi legati al modello relazionale. Una di queste è rappresentata proprio dalla tecnologia *Blockchain*.

La Blockchain è stata introdotta nel 2008, di pari passo con la criptovaluta Bitcoin, dal fondatore *Satoshi Nakamoto* (pseudonimo dell'inventore di tale tecnologia). Ha portato alla luce un'alternativa di archiviazione dei dati decentralizzata e cifrata.

Dall'inglese *"Catena di blocchi"*, la Blockchain si basa su una struttura a blocchi concatenati (chain). Ciò che caratterizza questa tecnologia è il legame crittografico tra i blocchi. Ogni blocco contiene al suo interno due hash: l'hash dei propri dati (che lo identifica univocamente) e l'hash del blocco precedente, che di fatto li concatena. Un hash è una stringa di lunghezza fissa calcolata da una funzione crittografica deterministica (a due input identici corrisponde uno stesso output). Questa struttura a catena garantisce due proprietà fondamentali: la **tracciabilità** e l'**immutabilità** dei dati registrati. Ogni tentativo di modifica di un blocco invalida l'intera catena successiva. Il vero punto di forza sta nel fatto che ogni manomissione risulta immediatamente rilevabile. Inoltre, in questo modo è possibile percorrere a ritroso l'intera cronologia delle modifiche fino al blocco di origine, chiamato *Genesis Block*.

L'unione della tecnologia Blockchain alla struttura tipica dei DBMS relazionali fornisce una potenziale alternativa ai metodi di archiviazione centralizzati per il trattamento dei dati sensibili.

[1]

[1]Il codice sorgente del prototipo sviluppato in questo lavoro di tesi, basato sull'approccio ibrido qui descritto, è disponibile per la consultazione su GitHub: `https://github.com/francescogallii/Blockchain-Decentralized-DB.git`

Capitolo 2

Modello relazionale

2.1 Introduzione

Proposto in una pubblicazione scientifica del 1970 da Edgar F. Codd, ricercatore dell'IBM a San Jose, California[4], il Modello Relazionale rappresenta il cuore del sistema di organizzazione dei dati dei database relazionali. Questo modello ha avuto una lenta affermazione a causa dell'alto livello di astrazione e della significativa richiesta computazionale dell'epoca. Per questo motivo non è stato possibile individuare realizzazioni efficienti rispetto alle strutture presenti sul mercato fino all'inizio degli anni 80.

Il modello relazionale presenta una notevole robustezza teorica e ha rivoluzionato la rappresentazione interna dei dati, permettendo la tutela degli utenti delle grandi banche dati. Il modello basa l'accesso ai dati sulla logica e non sulla loro posizione fisica. Per questa ragione sono state superate le limitazioni dei modelli dell'epoca, in particolare la **dipendenza dal percorso di accesso** (*access path dependence*) ai dati[4]. L'aspetto rivoluzionario sta nella possibilità di modificare frequentemente la struttura di memorizzazione senza cambiare modalità di interrogazione dei dati.

I precedenti modelli (gerarchico o reticolare) imponevano dei vincoli strettamente legati all'organizzazione fisica, rendendo più complessa l'evoluzione e gestione dei dati. La principale differenza è posta nel modo in cui vengono rappresentati i legami tra le diverse strutture dati. Il modello relazionale si basa esclusivamente sull'uso di valori o attributi comuni, mentre i precedenti modelli richiedevano l'utilizzo di percorsi di navigazione predefiniti o puntatori fisici, esponendo così la logica applicativa a future modifiche della struttura interna[4].

Favorendo la proprietà di **indipendenza dei dati** (considerata già all'epoca fondamentale),[4] il modello di base rappresenta questi ultimi come un insieme di **tabelle**, ciascuna composta da **tuple** (righe) e **attributi** (colonne).

2.2 La Struttura Formale

Il concetto matematico di relazione sta alla base del modello[4] ed è stato acquisito dalla teoria degli insiemi. La presenza di questi fondamenti concettuali ha determinato il successo di questo modello.

2.2.1 Relazione matematica

Si considerino n insiemi $D_1, D_2, \ldots, D_n$, non necessariamente distinti, chiamati **domini**, si definisce **prodotto cartesiano**

$$D_1 \times D_2 \times \cdots \times D_n$$

l'insieme di tutte le n-ple ordinate

$$(d_1, d_2, \ldots, d_n),$$

tali che d_i è un elemento del rispettivo dominio D_i. Una **relazione matematica** su quei domini è un qualunque sottoinsieme del prodotto cartesiano mostrato in precedenza[4;12].

Le principali proprietà di una relazione matematica sono[4;12]:

- **Grado**: numero n di domini coinvolti;
- **Cardinalità**: numero di n-ple che compongono la relazione;
- **Assenza di duplicati**: la relazione non può contenere n-ple identiche (essendo un insieme);
- **Assenza di ordine tra tuple**: l'ordine delle n-ple non è significativo;
- **Ordine dei domini**: l'ordine dei domini di una n-pla è significativo.

2.3 Dal Formalismo Matematico al Modello di Codd

Il modello relazionale introduce il concetto di **attributo**, dove a ogni dominio di una relazione viene associato un nome univoco, che ne specifica il ruolo. In questo modo viene superata l'ambiguità delle relazioni definite su domini non distinti. Le

n-ple di una relazione sono distinte una dall'altra, in quanto elementi di un insieme. La relazione nel modello relazionale non è più posizionale, in quanto l'ordine delle colonne (attributi) diventa irrilevante. Le componenti fondamentali, così come standardizzate dalla letteratura[12], sono:

- **Attributo**: Il nome associato a un dominio, che ne specifica il ruolo nella relazione (es. "Nome", "Matricola"). Nella pratica è inteso come l'intestazione di una colonna.
- **Dominio**: L'insieme dei valori ammissibili per un determinato attributo (es. l'insieme di tutti i possibili numeri di matricola).
- **Tupla**: Una n-pla di valori presi dai rispettivi domini. Si tratta in sostanza di una riga di una tabella o relazione, cioè l'insieme dei valori assunti dagli attributi specificati in cima a ciascuna colonna.
- **Schema di Relazione (o Intestazione)**: Il nome della relazione seguito dall'insieme dei suoi attributi (es. STUDENTE(Matricola, Nome, Cognome)).
- **Istanza di Relazione**: L'insieme delle tuple presenti in una relazione in un dato momento. È il contenuto effettivo della tabella, che varia nel tempo.
- **Database Relazionale**: Un insieme di schemi di relazione con nomi distinti.

2.4 Indipendenza dei Dati

Come espresso da Codd nella sua pubblicazione originale del 1970, l'obiettivo primario del modello relazionale è "proteggere i futuri utenti delle grandi banche dati dalla necessità di conoscere come i dati sono organizzati fisicamente sul sistema (la rappresentazione interna)".[4]

Si distinguono due livelli di indipendenza[4;12]:

- **Indipendenza fisica**: Un RDBMS presenta una separazione tra struttura logica e fisica. In questo modo il livello fisico non compromette l'accesso ai dati. Questo ottimizza le operazioni senza compromettere il software.
- **Indipendenza logica**: Permette di accedere ai dati logici indipendentemente dalla loro rappresentazione fisica. Quest'ultima può cambiare senza che i metodi di accesso ai dati logici debbano essere modificati.

I modelli precedenti, richiedendo riferimenti espliciti (puntatori) alla struttura fisica, non garantivano questo livello di astrazione.

2.5 Progettazione Logica: dal Modello E/R al Relazionale

L'implementazione di un database relazionale parte dal modello concettuale di Entità/Relazione (E/R). Il processo di traduzione (mapping) di questo schema in uno schema logico relazionale segue regole precise[12]. La regola base prevede che:

- Ogni **entità** (es. STUDENTE) diventi una **tabella** (relazione) nello schema logico.
- Gli **attributi** dell'entità (es. Nome, Cognome) diventino gli **attributi** (colonne) della tabella.
- L'**identificatore** (chiave) dell'entità diventi la **chiave primaria** (Primary Key) della tabella.

La traduzione delle associazioni tra entità, invece, dipende dalla loro cardinalità.

2.5.1 Traduzione delle associazioni tra tabelle

Vi sono delle regole di traduzione che vanno applicate a seconda della cardinalità dell'associazione tra due entità[12].

- **Associazione 1:1** (Uno-A-Uno): Rappresenta la relazione più semplice tra due tabelle. Ad un record di una delle due tabelle ne corrisponde uno dell'altra tabella e viceversa. Questa associazione viene gestita propagando la chiave primaria di una delle due entità come chiave esterna (Foreign Key) nella tabella associata. Per garantire la cardinalità 1:1, va aggiunto un vincolo di unicità (UNIQUE) a questa chiave esterna.
- **Associazione 1:M** (Uno-A-Molti): Ad un record della tabella del lato "1", possono corrispondere molteplici record nella tabella del lato "M". La chiave primaria della prima relazione verrà usata come chiave esterna nella seconda entità.
- **Associazione M:M** (Molti-A-Molti): La relazione più complessa tra due tabelle. Ad ogni record di una delle due tabelle possono corrispondere più record della seconda tabella. Non è rappresentabile direttamente nel modello relazionale, ma viene risolta creando una terza **tabella associativa**, che contiene le chiavi esterne verso entrambe le entità. Generalmente la chiave primaria di questa nuova entità è individuata dall'unione delle due Foreign Key.

2.6 Criticità nell'Architettura DBMS Centralizzata

Il modello relazionale presenta una notevole efficienza e maturità, ma la modalità di implementazione più comune avviene tramite architetture DBMS fortemente centralizzate. Questa centralizzazione, come anticipato nell'Introduzione, presenta delle vulnerabilità che il modello logico da solo non può risolvere.

Nel Database Administrator (DBA) e nell'integrità del server che ospita il DBMS è riposta l'intera fiducia del sistema. In questo modo viene strutturato il sistema intorno ad un **single point of failure** e, soprattutto per i **dati sensibili**, un *single point of trust*.

Le principali minacce in questo contesto, ampiamente trattate nei capitoli sulla sicurezza dei database[12], sono:

- **Manipolazione Interna**: Un amministratore malevolo o un utente con privilegi elevati può alterare o cancellare dati (es. log di accesso, transazioni finanziarie) senza lasciare tracce evidenti, compromettendo l'integrità.
- **Attacco Esterno Mirato**: Un attacco che riesce a violare il server centrale ottiene accesso all'intero patrimonio informativo.
- **Mancanza di Trasparenza**: è impossibile per un utente esterno verificare se un dato è stato modificato dopo la prima registrazione.

Come vedremo nei capitoli successivi, la tecnologia Blockchain offre un approccio alternativo per garantire le proprietà di tracciabilità e immutabilità dei dati. In questo modo verranno trattate le vulnerabilità tipiche di un DBMS centralizzato, proponendo un sistema ibrido per la tutela nel trattamento dei dati.

Capitolo 3

Introduzione alla tecnologia Blockchain

L'articolo che ha segnato la notorietà della tecnologia Blockchain è intitolato *Bitcoin: A Peer-to-Peer Electronic Cash System*[10] e risale al 2008, pubblicato dallo pseudonimo Satoshi Nakamoto. Nonostante questo, i suoi fondamenti concettuali risalgono a diversi anni prima.

Il primo concetto di *blind signatures* (firme cieche) per i pagamenti digitali non tracciabili risale al 1983, ad una pubblicazione del ricercatore David Chaum[3]

Tuttavia, il contributo più rilevante alla struttura della Blockchain proviene dai ricercatori Haber e Stornetta (1991), che proposero un metodo per apporre una marca temporale (timestamp) a documenti digitali, concatenandoli crittograficamente in modo da renderli immuni da retrodatazione o manomissioni[7;13].

La Blockchain (dall'inglese catena di blocchi) costituisce un registro digitale (o ledger), in cui i dati sono aggregati in 'Blocchi' concatenati cronologicamente[14]. La sua caratteristica fondamentale che separa questa tecnologia dai comuni database centralizzati è la distribuzione: questo registro non risiede in un'unica istanza centrale, ma è "gestito da un gruppo di nodi"[14] ed è replicato attraverso una rete *Peer-to-Peer* (P2P), dove ogni partecipante conserva in maniera totale o parziale una copia dei dati[14].

L'aspetto realmente rivoluzionario di questa tecnologia non sta nel semplice tracciamento delle transazioni (svolto già dai database tradizionali), ma nel concetto di fiducia. I sistemi convenzionali si fondano su terze parti fidate (trusted third parties), come le istituzioni finanziarie, per elaborare transazioni[10]. La Blockchain, invece, si propone come un abilitatore per "transazioni guidate dalla verifica (verification-

driven) tra parti che non hanno completa fiducia reciproca" [14]. Introduce un sistema basato su "prova crittografica (cryptographic proof) anziché sulla fiducia" [10], permettendo a due parti di negoziare direttamente ed eliminando la dipendenza da un intermediario centrale.

3.1 Proprietà Fondamentali

L'architettura della Blockchain introduce un insieme di proprietà intrinseche che ne definiscono il valore e la differenziano radicalmente dai sistemi tradizionali.

3.1.1 Decentralizzazione

Non esiste un singolo punto di controllo (né un single point of failure). La gestione della rete è demandata ai nodi stessi, che operano seguendo le regole definite dal protocollo. Ogni nodo possiede una replica del registro, garantendo un'elevata ridondanza, resilienza e resistenza alla censura.

3.1.2 Immutabilità

I dati registrati sulla Blockchain non sono modificabili. Questa proprietà, definita tamper-proof, assicura che un blocco, una volta validato e aggiunto, non possa essere alterato[13]. Ogni blocco è crittograficamente saldato al precedente; qualsiasi tentativo di manomissione retroattiva "invaliderebbe l'intera catena successiva" [10], rendendo l'attacco palese ed estremamente costoso sotto il punto di vista computazionale (in quanto richiederebbe nuovamente il calcolo proof-of-work di tutti i blocchi successivi a quello modificato). Questa caratteristica garantisce la provenienza (provenance) dei dati[14].

3.1.3 Sicurezza

La robustezza del registro è garantita da un duplice meccanismo: primitive crittografiche (per l'autenticazione) e un modello di consenso distribuito (per la validazione). Il sistema è concepito per essere sicuro finché la "maggioranza della potenza di calcolo (CPU) è controllata da nodi onesti" che non cooperano per attaccare la rete[10].

3.1.4 Trasparenza (e Tassonomia Pubblico/Privato)

Questa proprietà richiede un distinguo fondamentale, cruciale per il contesto di questa tesi. Nelle blockchain pubbliche (come Bitcoin o Ethereum), la trasparen-

za è assoluta: chiunque può partecipare alla rete e visionare l'intero storico delle transazioni.

Tuttavia, per la gestione di dati sensibili e per applicazioni aziendali, questo modello è inadeguato. Si impiegano, pertanto, architetture private o federate (consortium)[13]. In queste reti, definite permissioned, l'accesso è controllato: i partecipanti "sono identificati" e le loro attività sono "visibili solo a coloro che hanno la necessità di sapere" (need to know)[14], garantendo così la riservatezza richiesta e limitando la trasparenza al solo perimetro dell'organizzazione o del consorzio.

3.2 Architettura di Base: Nodi e Blocchi

Per comprendere il funzionamento della tecnologia, è necessario analizzare i suoi due componenti architetturali primari: i nodi che compongono la rete e i blocchi che formano il registro.

3.2.1 I Nodi e la Rete P2P

Un nodo è un partecipante alla rete (tipicamente un computer) che esegue il software della Blockchain e conserva una replica (completa o parziale) del registro. I nodi comunicano tra loro direttamente in una rete P2P per scambiare informazioni su "nuove transazioni" e "nuovi blocchi"[10].

La natura distribuita della rete è ciò che realizza la decentralizzazione. Affinché la rete mantenga una versione unica e condivisa della "verità", i nodi devono raggiungere un accordo sullo stato del registro. Questo processo è noto come consenso distribuito, ed è l'elemento chiave che permette alla rete di operare senza un'autorità centrale. Gli algoritmi che gestiscono questo accordo (come il Proof-of-Work) sono il cuore della sicurezza della Blockchain[13].

3.2.2 I Blocchi e la Catena

Un blocco è l'aggregato atomico del registro, un contenitore di dati composto da un corpo (l'elenco delle transazioni) e un'intestazione (header)[13].

Oltre al corpo, ogni blocco possiede un'intestazione (header). L'header contiene metadati fondamentali che definiscono l'identità e la posizione del blocco nella catena[13]. Tra questi, i più importanti per questa analisi sono:

- L'Hash del Blocco Precedente: Questo è il "collante" crittografico della catena. Includendo l'hash (un'impronta digitale univoca) del blocco che lo

precede, si crea un legame indissolubile che garantisce l'ordine cronologico e l'immutabilità[10].

- Il Timestamp: La marca temporale che certifica il momento della creazione del blocco[13].
- La Merkle Root: "Un singolo hash che riassume l'intero set di transazioni" [13], ottenuto tramite una struttura ad Albero di Merkle, che permette una verifica efficiente dei dati senza dover processare l'intero blocco[10].

La struttura dell'header, in particolare il riferimento all'hash del blocco precedente, garantisce la linearità e la storia della catena.

Tuttavia, questa architettura decentralizzata solleva due questioni fondamentali:

- Validità (Il problema del Double-Spending): Come fa la rete a sapere che le transazioni incluse nel blocco sono legittime? In un sistema privo di un arbitro centrale, come può un beneficiario "verificare che uno dei proprietari non abbia speso due volte (double-spend) l'asset?" [10].
- Creazione (Il problema del Consenso): In una rete decentralizzata, chi ha il diritto di raccogliere le transazioni e "creare" il prossimo blocco da aggiungere alla catena?

La risposta a entrambe le domande risiede nei meccanismi di validazione e consenso. Come accennato, i nodi devono raggiungere un accordo su una versione unica della verità. Questo processo è progettato per rendere la creazione di un blocco un'attività intenzionalmente difficile.

Per raggiungere questo obiettivo, l'header del blocco contiene anche altri metadati cruciali, come il Nonce (un numero arbitrario) e il target di Difficoltà[13], che sono gli strumenti tecnici utilizzati dagli algoritmi di consenso, uno dei quali è il Proof-of-Work[10].

Sarà oggetto del prossimo capitolo analizzare nel dettaglio questi processi:

- Il ruolo delle funzioni di hash crittografico nel garantire l'unicità e la sicurezza.
- Il processo di validazione delle transazioni basato sulla crittografia asimmetrica.
- Il funzionamento degli algoritmi di consenso (in particolare il mining) che utilizzano il Nonce e la Difficoltà per proteggere la rete e aggiungere nuovi blocchi.

Capitolo 4

Meccanismi di Sicurezza e Consenso

Nel capitolo precedente è stata introdotta l'architettura Blockchain, rappresentata come una catena di blocchi decentralizzata e immutabile. È stato trattato in maniera introduttiva come questa tecnologia garantisca un ordine cronologico senza un'autorità centrale, ma non è stato approfondito a livello pratico e crittografico.

Nei sistemi centralizzati la fiducia è riposta in un'autorità come il DBA (Database Administrator) che si occupa della gestione, archiviazione, accessibilità e ottimizzazione del database. Questa figura nel contesto della Blockchain non è presente e ci si basa unicamente su prove crittografiche e un consenso computazionale.

In questo capitolo verranno trattati i pilastri tecnici che rendono la Blockchain una valida alternativa ai sistemi centralizzati:

- Le funzioni di hash (integrità)
- Le firme digitali (validazione)
- La Proof-of-work (consenso)

4.1 Le Funzioni di Hash Crittografico

Il mattone fondamentale della Blockchain è la funzione di hash crittografico. Un hash (o impronta digitale) è l'output di un algoritmo matematico che trasforma una quantità arbitraria di dati in una stringa alfanumerica di lunghezza fissa[13]. Nel contesto di Bitcoin, l'algoritmo più utilizzato è lo SHA-256 (Secure Hash Algorithm 256-bit).

Per essere crittograficamente sicura, una funzione di hash deve possedere delle proprietà fondamentali:

- **Resistenza alla Preimmagine (Invertibilità):** È computazionalmente impossibile risalire ai dati di input originali (la "preimmagine") a partire solo dal suo hash. È una funzione a senso unico.
- **Resistenza alla Collisione:** È computazionalmente impraticabile trovare due input diversi che producano lo stesso identico hash.
- **Effetto Valanga (Avalanche Effect):** Qualsiasi minima modifica ai dati di input (anche un singolo bit) produce un hash di output completamente diverso e imprevedibile.
- **Deterministica**: Lo stesso input produrrà sempre lo stesso identico output. Questa proprietà è essenziale per la verifica: chiunque può ricalcolare l'hash di un blocco e verificare se corrisponde a quello memorizzato.
- **Lunghezza Fissa dell'Output**: Indipendentemente dalla dimensione dell'input (un singolo carattere o un'intera tabella di database), l'hash prodotto avrà sempre la stessa lunghezza (es. 256 bit per SHA-256). Questo standardizza l'archiviazione degli hash sulla blockchain.

Queste proprietà rendono l'hashing uno strumento ideale per certificare l'integrità dei dati: se l'hash di un documento oggi è lo stesso di domani, si ha la certezza crittografica che il documento non è stato alterato.

4.2 Applicazione degli Hash: Integrità della Catena

Nel capitolo precedente abbiamo visto che i blocchi contengono l'hash del blocco precedente e una "Merkle Root". Entrambi sono applicazioni dirette delle funzioni di hash.

4.2.1 Il Concatenamento Crittografico

L'intestazione (header) di ogni blocco contiene l'hash del blocco che lo precede. Questo semplice meccanismo crea la "catena" e garantisce l'immutabilità[10].

Si consideri un tentativo di manomissione di un blocco N:

- Un attaccante modifica una transazione nel blocco N.
- A causa dell'effetto valanga, l'hash del blocco N cambia completamente.

- Il blocco N+1, che memorizzava l'hash originale (e ora errato) del blocco N, diventa invalido.
- Per rendere valida la sua modifica, l'attaccante dovrebbe ricalcolare l'hash (e la Proof-of-Work) del blocco N, poi del blocco N+1, N+2, e così via, fino a superare l'intera catena onesta.

Questo concatenamento trasforma la modifica di un dato passato da un'operazione banale (come un UPDATE in un SQL centralizzato) a un'impresa computazionalmente insostenibile.

4.2.2 L'Albero di Merkle (Merkle Tree)

Un blocco non contiene un hash per ogni transazione nel suo header, ma raggruppa gli hash di tutte le transazioni utilizzando una struttura ad albero chiamata Merkle Tree[10].

Le transazioni (le "foglie" dell'albero) vengono sottoposte ad hash. Gli hash vengono poi accoppiati e sottoposti ad hash insieme, salendo di livello nell'albero fino a quando non rimane un unico hash: la Merkle Root.

Questo singolo hash, incluso nell'header del blocco, agisce come un'impronta digitale aggregata che certifica l'integrità di tutte le transazioni in quel blocco. Permette inoltre una verifica efficiente: per dimostrare che una transazione è inclusa in un blocco, è sufficiente fornire solo il "ramo" dell'albero che la collega alla radice, senza dover scaricare l'intero blocco.

Nella struttura progettuale che approfondiremo nei prossimi capitoli, si è scelto di non raggruppare gli hash di un sottogruppo di dati, poiché non si parla di transazioni, ma di campi di testo contenenti dati sensibili da salvaguardare e decifrabili mediante chiave privata.

4.3 Validazione delle Transazioni: La Firma Digitale

Prima che una transazione venga inclusa in un blocco, deve essere validata. Se l'hashing garantisce l'integrità dopo la registrazione, la crittografia asimmetrica garantisce la proprietà e l'autenticità al momento della creazione.

Come introdotto nel capitolo precedente, ogni utente possiede una coppia di chiavi:

- Una **Chiave Privata** (sk): Un dato segreto, noto solo al proprietario.

- Una **Chiave Pubblica** (pk): Derivata dalla chiave privata, è pubblica e funge da "indirizzo" per ricevere fondi o dati.

Quando un utente (es. Alice) vuole inviare una transazione, la crea e la "firma" utilizzando la propria chiave privata. Questa firma digitale è un sigillo crittografico che prova due cose:

- Autenticazione: Solo il possessore della chiave privata di Alice avrebbe potuto creare quella firma.
- Non Ripudio e Integrità: La firma è legata al contenuto specifico della transazione. Se la transazione venisse alterata (es. l'importo o il destinatario), la firma non sarebbe più valida.

Qualsiasi nodo della rete può quindi prendere la chiave pubblica di Alice (che è nota) e verificare la validità della firma. Se la verifica ha successo, il nodo è certo che la transazione è stata autorizzata da Alice.

Oltre alla firma, i nodi controllano anche il double-spending: verificano che i fondi (o input) che Alice sta tentando di spendere non siano già stati spesi in transazioni precedenti (in Bitcoin, questo avviene tramite il modello UTXO - Unspent Transaction Output).

4.4 Il Consenso: Proof-of-Work (PoW) e Mining

Abbiamo stabilito come validare le singole transazioni, ma non come la rete decentralizzata decida quali transazioni e in quale ordine inserirle nel prossimo blocco. Questo è il ruolo dell'algoritmo di consenso. Il Proof-of-Work (PoW) è la soluzione introdotta da Nakamoto per risolvere questo "concorso" per la creazione di blocchi in un ambiente P2P [10].

Il processo di PoW, noto come mining, è una competizione computazionale. I nodi specializzati (chiamati miners) eseguono i seguenti passaggi:

- Raccolta: Raccolgono le transazioni valide in attesa dalla rete (la mempool).
- Costruzione: Costruiscono un "blocco candidato", calcolando la Merkle Root e includendo l'hash del blocco precedente.
- Il Puzzle: Devono trovare un hash per il loro blocco candidato che soddisfi un requisito specifico, noto come Difficoltà (Difficulty).

Il puzzle non è altro che un'applicazione "brute force" della funzione di hash. L'header del blocco contiene un campo speciale chiamato Nonce (un numero usato una

sola volta). I miners modificano iterativamente questo Nonce e ricalcolano l'hash dell'intero header del blocco, milioni di volte al secondo.

Hash(Header del Blocco + Nonce) = Risultato

L'obiettivo è trovare un Nonce tale per cui il Risultato (l'hash del blocco) sia inferiore a un determinato valore Target. Visivamente, questo si traduce nel trovare un hash che inizi con un numero predeterminato di zeri (es. 0000000000000000000abc...).

Il primo miner che trova un hash valido (un "lavoro" o proof-of-work) vince il "concorso". A quel punto:

- Trasmette il suo blocco (e il Nonce vincente) alla rete.
- Gli altri nodi verificano (molto rapidamente, con un singolo calcolo di hash) che il blocco sia valido.
- Se valido, lo aggiungono alla propria copia della catena e iniziano a lavorare sul blocco successivo, utilizzando l'hash del blocco appena aggiunto come "hash precedente".

Il PoW risolve il problema del consenso "un-CPU-un-voto" [10]. Rende la creazione di blocchi difficile e costosa (in termini di energia elettrica e hardware), ma la verifica banale. Questo incentiva i miner a seguire le regole, poiché sprecare energia per creare un blocco invalido (che la rete rifiuterebbe) è economicamente svantaggioso.

Capitolo 5

La fusione tra RDBMS e Blockchain

5.1 Introduzione: La Convergenza Tecnologica

Nei capitoli precedenti sono state analizzate separatamente le caratteristiche principali dei database relazionali e della tecnologia blockchain. Queste tecnologie mostrano due visioni notevolmente distinte, se non addirittura opposte: i database relazionali (RDBMS) rappresentano uno standard per l'efficiente gestione dei dati centralizzati, mentre la tecnologia blockchain si pone come soluzione decentralizzata e immutabile.

Tuttavia, con l'evoluzione delle esigenze aziendali, finanziarie o sanitarie, verso modelli che richiedono sia alte prestazioni che garanzie di auditabilità, è emersa un'ulteriore alternativa che implica la convergenza di queste tecnologie apparentemente opposte. In letteratura viene evidenziato un forte potenziale nella possibilità di "sfruttare le ricche funzionalità e le capacità di elaborazione transazionale già integrate nei database relazionali in decenni di ricerca e sviluppo" per strutturare sistemi blockchain più robusti e integrati[11].

Questo capitolo analizza l'integrazione tra blockchain e database relazionali, ovvero la fusione tra le proprietà di un registro distribuito mappate su un'architettura relazionale esistente.

5.2 Analisi delle lacune e dei punti di forza

Sebbene queste due tecnologie siano apparentemente diverse, in particolare nel loro "modello di fiducia" (trust model), esistono numerose similitudini funzionali tra le piattaforme blockchain distribuite e i database relazionali replicati.

Molte piattaforme blockchain esistenti tendono a ricostruire da zero funzionalità che i database possiedono da decenni, trattando il database sottostante solo come un mero archivio di informazioni. L'articolo *Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database* mostra una visione contraria, suggerendo che un approccio migliore sia quello di arricchire i DBMS esistenti con proprietà crittografiche[11].

La Tabella 5.1 riassume le principali differenze e gli adattamenti necessari per trasformare un database relazionale in una blockchain permissioned, basandosi sull'analisi svolta dai ricercatori di IBM.

Proprietà Blockchain	**Funzionalità RDBMS**	**Miglioramento Necessario**
Smart Contract	Stored Procedures (PL/SQL)	Esecuzione deterministica
Autenticità e Non Ripudio	Gestione Utenti	Firme digitali sulle transazioni
Immutabilità	Transaction Logs	Log concatenati crittograficamente
Consenso Decentralizzato	Replicazione Master-Slave	Consenso tra nodi non fidati

Tabella 5.1: Confronto tra proprietà Blockchain e Database Relazionali (adattato da[11])

5.2.1 Dagli Smart Contract alle Stored Procedures

Nelle blockchain come Ethereum o Hyperledger Fabric, gli smart contract sono funzioni che gestiscono lo stato del ledger. Un parallelo diretto nei database relazionali è rappresentato dalle *Stored Procedures* (es. PL/SQL). Tuttavia, mentre le stored procedures standard possono contenere funzioni non deterministiche (come `random()` o `timestamp`), in un contesto blockchain è essenziale "vincolare le procedure PL/SQL per garantire che l'esecuzione sia deterministica"[11], affinché tutti i nodi presentino lo stesso stato.

5.2.2 Autenticità e Non Ripudio

I database tradizionali gestiscono l'accesso tramite autenticazione utente/password, ma le transazioni, una volta registrate, non portano con sé una prova crittografica di chi le ha generate. Al contrario, in un sistema blockchain, "le transazioni sono firmate digitalmente dall'invocatore e registrate, rendendole non ripudiabili" [11]. Integrare questa proprietà in un RDBMS richiede che ogni operazione di scrittura sia accompagnata dalla firma digitale del client.

5.2.3 Il Problema del Consenso in Ambienti Non Fidati

La differenza architetturale più profonda risiede nel modello di replicazione. I database distribuiti utilizzano da tempo protocolli di replicazione (Master-Slave o Master-Master) per garantire la disponibilità dei dati.

Il limite di questi protocolli classici risiede nel fatto che "il nodo master è considerato fidato in quanto è l'unico che esegue le transazioni e propaga gli aggiornamenti finali agli altri master e slave" [11]. Questo modello fallisce in un contesto blockchain, dove le organizzazioni partecipanti (i nodi) si conoscono ma sono "mutuamente diffidenti" (mutually distrustful).

In una rete decentralizzata, un nodo non può fidarsi ciecamente dei risultati dell'esecuzione di un altro nodo. Di conseguenza, "tutte le transazioni devono essere eseguite indipendentemente su tutti i nodi" [11]. La sfida tecnica principale diventa quindi garantire che tale esecuzione indipendente porti allo stesso stato serializzato su tutti i replicati, senza un coordinatore centrale fidato.

5.3 Architetture di Elaborazione Transazionale

Per risolvere il problema del consenso e dell'ordinamento delle transazioni in un database blockchain, la letteratura[11] identifica due approcci architetturali principali.

5.3.1 Approccio Order-then-Execute

In questo modello, tipico di molte piattaforme blockchain, le transazioni vengono prima ordinate attraverso un servizio di consenso e raggruppate in blocchi. Successivamente, i nodi "eseguono le transazioni in sequenza nell'ordine in cui appaiono nel blocco per garantire la serializzabilità e la coerenza tra i nodi" [11].

Questo approccio è più semplice da implementare e garantisce che tutti i nodi processino gli stessi dati nello stesso ordine. Tuttavia, l'esecuzione sequenziale può limitare il throughput del sistema.

5.3.2 Approccio Execute-Order-in-Parallel

Un approccio più innovativo proposto da Nathan et al. prevede che l'esecuzione delle transazioni e il loro ordinamento avvengano in parallelo. In questo schema, i nodi eseguono le transazioni in isolamento (sfruttando tecniche come il *Serializable Snapshot Isolation* o SSI) e, parallelamente, un servizio di ordinamento stabilisce il consenso. Sebbene promettente in termini di prestazioni, questo approccio richiede modifiche sostanziali al motore del database per gestire conflitti di concorrenza in un ambiente decentralizzato[11].

5.4 Verso un'Architettura Ibrida Permissioned

Alla luce dell'analisi dello stato dell'arte, emerge che la creazione di una "Blockchain Relational Database" è una strada percorribile che permette di mantenere i vantaggi di interrogazione complessa (SQL) e la gestione dei dati tipica dei database, aggiungendo al contempo le proprietà di immutabilità e verifica decentralizzata.

Come affermato nello studio di riferimento, applicazioni che hanno "forti requisiti di conformità e audit e necessità di query analitiche ricche... sono quelle che trarranno maggior beneficio dal database relazionale blockchain" [11].

Questa tesi adotta e adatta questi concetti. Nel prossimo capitolo, verrà presentata l'architettura specifica progettata per questo lavoro, che segue un modello ibrido in cui il database relazionale funge da livello di persistenza e validazione, mentre la logica di "mining" e crittografia viene delegata al client, realizzando una variante del modello *Order-then-Execute* ottimizzata per la privacy dei dati sensibili.

Capitolo 6

Dalla Teoria alla Pratica: Architettura del Sistema Ibrido

6.1 Introduzione: L'Adattamento dei Paradigmi Blockchain

Nei capitoli precedenti sono stati introdotti i fondamenti del modello relazionale[4] e della tecnologia blockchain, quest'ultima analizzata primariamente attraverso il suo pioniere: Bitcoin[10]. I concetti di hashing, concatenamento crittografico, firme digitali e consenso distribuito (come il Proof-of-Work) rappresentano il cuore delle reti pubbliche e permissionless.

Tuttavia, l'applicazione di questa tecnologia a un contesto specifico — la gestione di dati sensibili — richiede un adattamento architetturale significativo[8]. Un sistema progettato per la trasparenza finanziaria assoluta (Bitcoin) è intrinsecamente inadatto a un dominio che esige privacy, confidenzialità e controllo degli accessi.

Questo capitolo funge da "ponte" tra la teoria generale e l'architettura pratica del nostro progetto. Si analizzeranno le differenze fondamentali tra il modello pubblico di Bitcoin e il modello ibrido e permissioned qui proposto. Come si vedrà, le differenze architetturali identificate non costituiscono una deviazione dai principi di sicurezza, ma rappresentano delle scelte progettuali, volte a ottimizzare il sistema per il trattamento sicuro dei dati, unendo i benefici di immutabilità della blockchain con la struttura e l'efficienza dei database relazionali[6;14].

6.2 Differenze Chiave e Punti di Forza dell'Architettura

L'architettura sviluppata in questo progetto si discosta dal modello Bitcoin [10] in diversi punti chiave, ognuno dei quali rappresenta un punto di forza per il caso d'uso specifico.

6.2.1 Dal Dato Finanziario al "Payload" Crittografato

La prima e più fondamentale differenza risiede nella natura dell'informazione registrata.

- Modello Bitcoin: L'unità atomica è la transazione finanziaria (un trasferimento di valore). Un blocco aggrega migliaia di queste transazioni. Per gestire tale volume in modo efficiente, si utilizza un Albero di Merkle per compattare l'hash di tutte le transazioni in un'unica "Merkle Root" [10].

- Architettura Proposta: L'unità atomica è un singolo "payload" di dati sensibili (il data_text fornito dal "Creator"). Ogni blocco nella nostra catena rappresenta la notarizzazione di un singolo record di dati.

Di conseguenza, come già anticipato nel Capitolo 4, l'utilizzo di un Albero di Merkle risulta superfluo. L'hash del blocco (block_hash) viene calcolato includendo direttamente l'hash dei dati crittografati, garantendo l'integrità in modo più diretto ed efficiente per la nostra struttura 1-a-1 (un blocco per un dato), un approccio ibrido che mira a rendere tamper-proof (inviolabili) i dati di un database relazionale [9].

6.2.2 Da "Permissionless" a "Permissioned": Il Ruolo dei *Creators*

Il modello di Bitcoin è permissionless: chiunque può partecipare alla rete [10]. Questo è inaccettabile per la gestione di dati sensibili [8].

Nell'architettura proposta il sistema è permissioned [1]. L'accesso e la possibilità di scrivere dati sono limitati a entità pre-autorizzate: i "Creators". Come visibile nel database (tabella blockchain.creators), ogni creator è identificato da una chiave pubblica RSA. Solo un utente in possesso della chiave privata corrispondente può firmare e sottomettere un nuovo blocco.

6.2.3 Il Nuovo Ruolo del Proof-of-Work: dal Consenso al "Mining Client-Side"

Questa è la differenza architetturale più significativa. Nelle blockchain pubbliche, il Proof-of-Work (PoW) è un meccanismo di consenso di rete per decidere chi, tra migliaia di "miner" in competizione, ha il diritto di creare il prossimo blocco[10].

Nel nostro sistema, il creatore del blocco è già noto (è il "Creator" che sottomette i dati). Il ruolo del PoW viene quindi ridefinito, tornando al suo scopo concettuale originale: una contromisura ai denial-of-service e allo spam[2;5].

- Spostamento sul Client: Il PoW non viene eseguito dai nodi della rete, ma direttamente dal client (nel browser dell'utente), come implementato nello script frontend/src/utils/cryptoUtils.js.
- Rate-Limiting e "Proof of Commitment": Lo scopo del PoW cambia. Non serve più per il consenso, ma agisce come meccanismo anti-spam (rate-limiting). Obbliga il client a spendere una quantità non banale di tempo computazionale (in base alla DIFFICULTY) per creare un blocco. Questo impedisce a un singolo creator di inondare la rete di blocchi e funge da "prova di impegno" (Proof of Commitment) che certifica lo sforzo computazionale al momento della creazione.
- Preservazione dell'Immutabilità: Mantiene la proprietà di immutabilità della catena. Qualsiasi tentativo di alterare un blocco storico richiederebbe il ricalcolo del PoW per quel blocco e per tutti i successivi, che è un'operazione computazionalmente proibitiva.

6.2.4 Nodi Validatori, non "Miner"

Dato che il mining avviene sul client, il ruolo dei nodi P2P (definiti nel docker-compose.yml come node1, node2, node3) cambia radicalmente.

- Modello Bitcoin: I nodi sono "miner" in competizione[10].
- Architettura Proposta: I nodi sono "Validatori" e "Replicatori" in collaborazione, un modello tipico dei sistemi ibridi e permissioned[6;1]. Il loro compito (implementato nel backend, es. routes/blocks.js) non è creare blocchi, ma:
  - Ricevere un blocco già minato e firmato dal client.
  - Verificare la sua validità: Il PoW (l'hash) è corretto? La firma digitale corrisponde alla chiave pubblica del Creator?
  - Salvare il blocco valido nel database PostgreSQL (tramite blockchain.js).

- Propagare il blocco valido agli altri nodi della rete (tramite p2p.js) per garantire la replica e la ridondanza.

6.2.5 Sicurezza e Privacy: Crittografia End-to-End

Infine, l'architettura risponde alla necessità di privacy assoluta per i dati sensibili.

- Modello Bitcoin: La trasparenza è la regola di base. Tutte le transazioni sono pubbliche e visibili a chiunque.
- Architettura Proposta: La privacy è by-design. Il sistema implementa una crittografia end-to-end (E2E) robusta, gestita interamente sul client. Questa è una delle tecniche fondamentali per la protezione dei dati sensibili nei sistemi basati su blockchain[8].
  - Il dato in chiaro (data_text) non lascia mai il browser dell'utente.
  - Viene generata una chiave simmetrica (AES-256-GCM) effimera per crittografare il dato.
  - La chiave AES stessa viene poi crittografata usando la chiave pubblica RSA (2048-bit) del Creator.
  - Il backend (blockchain.blocks) riceve e salva unicamente i dati cifrati (encrypted_data), la chiave AES cifrata (encrypted_data_key) e il vettore di inizializzazione (data_iv).

In questo modo, i nodi server e il database non possono in alcun modo leggere il contenuto dei dati sensibili, garantendo la massima confidenzialità. L'unico in grado di decifrare i dati è il Creator originale, fornendo la propria chiave privata (come implementato nella sezione DataDecryption.jsx).

6.3 Conclusione: Un Sistema Ibrido Ottimizzato

L'architettura qui proposta unisce il meglio di due mondi: l'efficienza interrogativa, la strutturazione dei dati e la maturità del modello relazionale[4] (implementato su PostgreSQL) e i principi di immutabilità, tracciabilità e decentralizzazione della fiducia della blockchain[13].

Sfruttando un modello permissioned[1], spostando il PoW sul client[2] e implementando una crittografia E2E[8], si ottiene un sistema che non si limita a copiare la blockchain, ma la adatta per creare un registro sicuro, privato e immutabile, specificamente progettato per il trattamento dei dati sensibili. I capitoli successivi descriveranno nel dettaglio l'implementazione di questi concetti.

Capitolo 7

Implementazione del Nodo Validatore e API

7.1 Introduzione

Nel capitolo precedente è stata introdotta l'architettura ibrida, illustrando le principali differenze tra una blockchain dedicata al tracciamento delle transazioni di criptovalute e una dedicata all'archiviazione di dati sensibili. Per lo sviluppo del server si è optato per Node.js e il framework Express. In questa architettura esso assume il ruolo cruciale di nodo validatore e replicatore, una funzione che nel contesto Bitcoin è invece svolta dal "miner".

Le sue responsabilità principali sono:

- Definire e inizializzare lo schema del database su PostgreSQL (livello di persistenza per la catena).
- Strutturare un'interfaccia API RESTful sicura per permettere l'interazione del client con il sistema.
- Gestire la logica di registrazione di nuovi "Creators" (**permissioning**).
- Validare e persistere i nuovi blocchi sottomessi dai client.
- Fornire ai client autorizzati i dati crittografati per la decifratura locale.

Questo capitolo approfondisce nel dettaglio la logica di backend, mettendo in evidenza come il modello relazionale sia stato adattato ai principi della blockchain.

7.2 Definizione dello Schema (db.js)

Il primo file che analizziamo è `backend/src/database/db.js`, che si occupa della definizione della struttura dati relativa ai creatori dei blocchi e alla catena stessa. Quest'ultimo utilizza la libreria `pg` (node-postgres) per connettersi al database PostgreSQL.

7.2.1 Definizione delle Tabelle

Le due tabelle principali sono `blockchain.creators` e `blockchain.blocks`, che traducono i concetti discussi in precedenza in *TABLE* dello schema relazionale. Il seguente codice mostra la definizione SQL della tabella `blockchain.creators`.

```
1  await client.query(`
2      CREATE TABLE IF NOT EXISTS blockchain.creators (
3          creator_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
4          display_name VARCHAR(255) UNIQUE NOT NULL CHECK (length(
               display_name) >= 3),
5          public_key_pem TEXT NOT NULL,
6          key_algorithm VARCHAR(50) NOT NULL DEFAULT 'RSA-OAEP',
7          key_size INTEGER NOT NULL DEFAULT 2048 CHECK (key_size >=
               2048),
8          is_active BOOLEAN DEFAULT TRUE,
9          created_at TIMESTAMPTZ DEFAULT NOW(),
10         updated_at TIMESTAMPTZ DEFAULT NOW()
11     );
12 `);
```

Listing 7.1: Definizione della tabella `blockchain.creators` in `db.js`

Successivamente viene illustrata la tabella `blockchain.blocks`, che definisce la catena di blocchi.

```
1  await client.query(`
2      CREATE TABLE IF NOT EXISTS blockchain.blocks (
3          block_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
4          block_number BIGSERIAL UNIQUE, -- HEIGHT COLONNA PER
               ORDINAMENTO E INTEGRITA'
5          creator_id UUID REFERENCES blockchain.creators(creator_id)
               ON DELETE SET NULL,
6          previous_hash VARCHAR(64),
7          block_hash VARCHAR(64) UNIQUE NOT NULL,
8          nonce BIGINT NOT NULL,
9          difficulty INTEGER NOT NULL CHECK (difficulty >= 1 AND
               difficulty <= 10),
10
11         -- Dati crittografati
12         encrypted_data BYTEA NOT NULL,
```

```
13          data_iv BYTEA NOT NULL,
14          encrypted_data_key BYTEA NOT NULL,
15          data_size INTEGER NOT NULL CHECK (data_size > 0),
16
17          -- Firma e verifica
18          signature BYTEA NOT NULL,
19          created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
20          verified BOOLEAN DEFAULT FALSE,
21          verified_at TIMESTAMPTZ,
22
23          mining_duration_ms INTEGER,
24
25          -- Controlla che il blocco genesi abbia previous_hash NULL
                o GENESIS_HASH e block_number = 1
26          -- Se block_number > 1, previous_hash non deve essere NULL
27          CONSTRAINT valid_genesis_block CHECK (
28              (previous_hash IS NULL AND block_number = 1) OR
29              (previous_hash IS NOT NULL AND block_number > 1)
30          )
31      );
32  `);
```

Listing 7.2: Definizione della tabella `blockchain.blocks` in `db.js`

L'analisi di queste due tabelle mostra le scelte implementative che caratterizzano questo modello:

- **blockchain.creators**: Questa *TABLE* implementa il livello di *permissioning*. L'univocità è garantita da `display_name`, mentre `public_key_pem` memorizza la chiave pubblica RSA necessaria per validare le firme dei blocchi sottomessi.

- **blockchain.blocks**: Questa è la catena di blocchi vera e propria. Si può notare l'uso di `BIGSERIAL` per `block_number`, che rappresenta l'*altezza* del blocco (posizione nella catena) e sfrutta una feature del database relazionale per garantire l'ordine e l'univocità dei blocchi, affiancandosi alla garanzia crittografica di `previous_hash`.

- **Payload Crittografato**: I campi `encrypted_data`, `data_iv` e `encrypted_data_key` sono di tipo `BYTEA` (array di byte). Il database memorizza solo il *ciphertext* senza conoscere il contenuto in chiaro, applicando il principio di "privacy by-design" discusso nel capitolo precedente.

- **Vincoli di Integrità**: Il `CONSTRAINT valid_genesis_block` impone a livello di database le regole strutturali della catena (il primo blocco non ha un predecessore).

7.2.2 Immutabilità a Livello Database

Oltre all'immutabilità crittografica, è stato implementato un meccanismo di protezione tramite un *trigger*. Questo impedisce `DELETE` o `UPDATE` non autorizzati delle istanze della tabella `blockchain.blocks`. Il codice seguente mostra la funzione `prevent_blockchain_tampering()` e il trigger `block_chain_protect` che la applica.

```
await client.query(`
    CREATE OR REPLACE FUNCTION prevent_blockchain_tampering()
    RETURNS trigger AS $$
    BEGIN
        -- permette UPDATE SOLO sul campo 'verified'
        IF TG_OP = 'UPDATE' THEN
            IF (OLD.verified IS DISTINCT FROM NEW.verified) THEN
                -- Permette l'aggiornamento solo se solo 'verified' cambia
                RETURN NEW;
            ELSE
                -- Proibisce qualsiasi altro UPDATE sui dati del blocco (hash, nonce, dati, ecc.)
                RAISE EXCEPTION 'block_chain is append-only: UPDATE prohibited, except for "verified" status.';
            END IF;
        ELSIF TG_OP = 'DELETE' THEN
            -- Proibisce DELETE
            RAISE EXCEPTION 'block_chain is append-only: DELETE prohibited.';
        END IF;
        -- Dovrebbe essere irraggiungibile per INSERT/TRUNCATE se non gestiti esplicitamente
        RETURN NEW;
    END;
    $$ LANGUAGE plpgsql;
`);

await client.query(`
    CREATE TRIGGER block_chain_protect
    BEFORE UPDATE OR DELETE ON blockchain.blocks
    FOR EACH ROW
    EXECUTE FUNCTION prevent_blockchain_tampering();
`);
```

Listing 7.3: Trigger per la protezione da manomissioni in `db.js`

Si può notare esplicitamente che è autorizzata solo la modifica del campo `verified` (gestito dal servizio di audit asincrono), ma viene bloccata qualsiasi altra modifica, rendendo la tabella *append-only* e rafforzando l'immutabilità della catena. Questa

caratteristica protegge il database da attacchi diretti (inclusi quelli da parte di un amministratore).

7.3 Gestione delle Identità: Registrazione dei "Creators" (creators.js)

Il file `backend/src/routes/creators.js` gestisce l'endpoint API per il *permissioning*. La sua funzione più importante è `POST /`, che permette la registrazione di un nuovo utente autorizzato a scrivere sulla blockchain. Il seguente codice illustra la logica di registrazione e validazione.

```
1  router.post('/', validateCreator, asyncHandler(async (req, res) =>
       {
2      const errors = validationResult(req);
3      if (!errors.isEmpty()) {
4          throw new ValidationError('Validation failed', errors.array
               ());
5      }
6
7      const { display_name, public_key_pem } = req.body;
8
9      // Validazione avanzata della chiave pubblica
10     const keyValidation = CryptoUtils.validatePublicKeyPem(
           public_key_pem);
11     if (!keyValidation.valid) {
12         throw new ValidationError('Invalid RSA public key',
               keyValidation.error);
13     }
14
15     try {
16         const insertQuery = `
17             INSERT INTO blockchain.creators (display_name,
                   public_key_pem, key_algorithm, key_size)
18             VALUES ($1, $2, $3, $4)
19             RETURNING creator_id, display_name, key_algorithm,
                   key_size, created_at`;
20
21         const { rows } = await pool.query(insertQuery, [
22             display_name,
23             public_key_pem,
24             keyValidation.keyType || 'RSA-OAEP', // Usa il tipo
                   validato
25             keyValidation.keySize // Usa la dimensione validata
26         ]);
27
```

```
28          logger.info('New creator registered', { creatorId: rows[0].
                creator_id, displayName: display_name });
29          res.status(201).json({ message: 'Creator registered
                successfully', creator: rows[0] });
30      } catch (e) {
31          if (e.code === '23505') { // PostgreSQL unique violation
                for display_name
32              logger.warn('Attempted to register duplicate creator
                    display name', { displayName: display_name });
33              throw new ConflictError('Display name already taken');
34          }
35          logger.error('Error registering creator', { error: e.
                message, stack: e.stack });
36          throw e; // Rilancia l'errore per il gestore globale
37      }
38  }));
```

Listing 7.4: Registrazione di un nuovo Creator in `creators.js`

Questo frammento di codice rappresenta il cuore del processo di validazione, dove vengono eseguiti i controlli fondamentali.

- Il server riceve il `display_name` e la `public_key_pem` dal client, che ha già generato la coppia di chiavi localmente.
- Viene eseguita una validazione (`CryptoUtils.validatePublicKeyPem`) per assicurare che la chiave pubblica ricevuta sia conforme agli standard di sicurezza (es. RSA 2048-bit).
- Se la validazione ha successo, il nuovo creator viene inserito nel database. Successivamente il sistema considererà valida la firma prodotta dalla chiave privata associata.

Infine l'endpoint `GET /:display_name/public-key` funge da rubrica pubblica, permettendo ai client di recuperare la chiave pubblica di un creator, necessaria per avviare il processo di cifratura, decifratura e sottomissione del blocco.

7.4 Il Cuore del Validatore: Creazione e Validazione dei Blocchi (blocks.js)

Questo è il file più importante del backend, poiché implementa la logica di validazione e persistenza dei blocchi. Come discusso nel capitolo precedente, il processo è suddiviso in due fasi per supportare il mining e la crittografia E2E lato client.

7.4.1 Fase 1: Preparazione al Mining (Client-Side)

Il client non può iniziare il mining finché non conosce l'hash del blocco precedente. L'endpoint `POST /blocks/prepare-mining` serve a fornire queste informazioni.

```
// POST /blocks/prepare-mining
router.post('/prepare-mining', validatePrepareMining, asyncHandler(
    async (req, res) => {
     const errors = validationResult(req);
     if (!errors.isEmpty()) {
         throw new ValidationError('Validation failed for mining
             preparation', errors.array());
     }

     const { display_name, data_text } = req.body; // Solo nome e
        dati grezzi

     // 1. Trova il Creator e la sua chiave pubblica
     const creatorResult = await pool.query(
         'SELECT creator_id, public_key_pem FROM blockchain.creators
             WHERE display_name = $1 AND is_active = true',
         [display_name]
     );
     if (creatorResult.rows.length === 0) {
         throw new NotFoundError('Creator not found or inactive');
     }
     const { creator_id, public_key_pem } = creatorResult.rows[0];

     // 2. Determina previousHash
     const lastBlock = await req.blockchain.getLatestBlock();
     const previousHashForCalc = lastBlock ? lastBlock.block_hash :
        GENESIS_HASH;

     logger.info(`Preparing mining for creator ${display_name}.
        Sending public key and previous hash.`);

     // 3. Invia la chiave pubblica e le info necessarie al frontend
     res.json({
         creator_id: creator_id,
         public_key_pem: public_key_pem,
         previous_hash: previousHashForCalc,
         difficulty: DIFFICULTY,
     });
}));
```

Listing 7.5: Endpoint per la preparazione al mining in `blocks.js`

Il server recupera la chiave pubblica del creator (necessaria al client per la crittografia E2E della chiave AES) e l'hash dell'ultimo blocco (`previousHashForCalc`).

Con queste informazioni il client ha tutto il necessario per crittografare i dati, eseguire il PoW e firmare l'hash del blocco.

7.4.2 Fase 2: Validazione e Commit del Blocco

Dopo che il client sarà riuscito a completare la PoW rispettando la difficoltà (in termini di zeri iniziali), si occuperà di inviare il blocco completo all'endpoint `POST /blocks/commit`, dove il nodo validatore eseguirà la verifica.

```
1  // POST /blocks/commit
2  router.post('/commit', validateCommitBlock, asyncHandler(async (req
       , res) => {
3      // ... validazione input ...
4      const {
5          creator_id, block_hash, signature_hex, ...
6      } = req.body;
7
8      // 1. Verifica l'esistenza del creator e recupera la chiave
           pubblica
9      const creatorResult = await pool.query(
10         'SELECT public_key_pem FROM blockchain.creators WHERE
               creator_id = $1 ...',
11         [creator_id]
12     );
13     // ... gestione errore ...
14     const { public_key_pem } = creatorResult.rows[0];
15
16     // 2. Verifica la firma digitale (Autenticita')
17     const signature = Buffer.from(signature_hex, 'hex');
18     const isSignatureValid = CryptoUtils.verifySignature(
19         public_key_pem, block_hash, signature);
20     if (!isSignatureValid) {
21         throw new BlockchainError('Invalid digital signature...');
22     }
23
24     // 3. Verifica Proof-of-Work (Impegno)
25     const requiredPrefix = '0'.repeat(difficulty);
26     if (!block_hash.startsWith(requiredPrefix)) {
27         throw new BlockchainError('Proof-of-Work validation failed
               ');
28     }
29
30     // 4. Determina previousHash per DB
31     const lastBlock = await req.blockchain.getLatestBlock();
32     const previousHashForDb = lastBlock ? lastBlock.block_hash :
           null;
33     const newBlockNumber = lastBlock ? BigInt(lastBlock.
           block_number) + 1n : 1n;
```

```
34
35      // 5. Prepara i dati per l'inserimento nel DB (converti hex in
            Buffer)
36      const newBlockData = { /* ... */ };
37
38      // 6. Aggiungi alla chain locale (DB + memoria) e trasmetti
39      const added = await req.blockchain.addBlock(newBlockData);
40
41      if (added) {
42            req.p2pServer.broadcastBlock(newBlockData); // Trasmetti
                 ai peer
43            logger.info(`Block #${newBlockData.block_number}
                 committed...`);
44      }
45
46      res.status(201).json({ message: 'Block committed and
            broadcasted', ... });
47  }));
```

Listing 7.6: Endpoint per il commit e la validazione del blocco in `blocks.js`

Il frammento di codice mostrato in precedenza rappresenta il cuore del processo di validazione, dove vengono eseguiti i controlli fondamentali:

1. **Verifica dell'Autenticità (Firma):** Il server usa la chiave pubblica del creator (recuperata dal DB) per verificare la firma (`isSignatureValid`). Questo garantisce che il blocco provenga da un utente autorizzato e che l'hash non sia stato alterato dopo la firma.

2. **Verifica della Proof-of-Work (impegno computazionale):** Viene verificato che l'hash del blocco rispetti la difficoltà prevista, ovvero che inizi con il numero di zeri richiesto (requiredPrefix). Questa condizione certifica che il client ha eseguito un adeguato lavoro computazionale, fungendo da meccanismo di rate limiting e di protezione contro la generazione massiva di blocchi.

3. **Persistenza e Propagazione:** Solo in caso di esito positivo di entrambe le verifiche, il blocco viene convertito in un formato compatibile con il database, salvato nella blockchain locale e successivamente trasmesso ai nodi della rete peer-to-peer, garantendo la sincronizzazione del registro distribuito.

7.5 Accesso Sicuro ai Dati: Endpoint di Decifratura (decrypt.js)

Il file `backend/src/routes/decrypt.js` completa il ciclo di vita del dato, dimostrando l'implementazione di un'architettura end-to-end (E2E) basata su principi di sicurezza e riservatezza.

```
// ** NUOVA ROTTA: GET /decrypt/blocks/:creator_id **
router.get(
    '/blocks/:creator_id',
    [ param('creator_id').isUUID().withMessage('Invalid Creator ID format') ],
    asyncHandler(async (req, res) => {
        // ... validazione input ...

        const { creator_id } = req.params;

        // ... recupero display_name ...

        // Recupera tutti i blocchi crittografati per quel creator
        const blocksResult = await pool.query(
            `SELECT
                block_id, block_number, block_hash, created_at,
                encode(encrypted_data, 'base64') as encrypted_data_b64,
                encode(data_iv, 'base64') as data_iv_b64,
                encode(encrypted_data_key, 'base64') as encrypted_data_key_b64,
                data_size, verified
             FROM blockchain.blocks
             WHERE creator_id = $1
             ORDER BY block_number ASC`,
            [creator_id]
        );

        // ...
        res.json({
            creator_id: creator_id,
            display_name: displayName,
            blocks: blocks // Invia i blocchi crittografati (con dati in base64)
        });
    })
);
```

Listing 7.7: Endpoint per il recupero dei dati crittografati in `decrypt.js`

La caratteristica fondamentale di questo endpoint è che non decifra nulla. Il server

recupera i dati richiesti dal database, li converte in un formato testuale adatto al trasferimento JSON e li invia al client. Quest'ultimo si occuperà della decifratura, che avviene nel browser dell'utente (l'unico a possedere la chiave privata). In questo modo è garantito che il server non possa accedere al contenuto sensibile, rispettando il requisito di *zero-knowledge* (dal punto di vista del server) del sistema.

Capitolo 8

Implementazione del Client: Crittografia E2E e Proof-of-Work Client-Side

8.1 Introduzione: Il "Client Pesante" Crittografico

Nei capitoli precedenti è stata definita l'architettura ibrida del sistema, che affida al nodo validatore (backend) le funzioni di persistenza, validazione e replica (Capitolo 6). Come delineato nel Capitolo 5, il vero cuore della sicurezza e della privacy del sistema non risiede nel server, bensì nel **client**.

L'applicazione frontend, sviluppata in React, non è una semplice interfaccia utente (UI), ma un vero e proprio "client pesante" crittografico. È progettata per eseguire in modo autonomo tutte le operazioni sensibili direttamente nel browser dell'utente, assicurando che i dati in chiaro e, soprattutto, le chiavi private dei "Creator",restino sempre all'interno del dispositivo locale.

In questo capitolo viene descritta nel dettaglio l'implementazione del frontend, con particolare attenzione a tre flussi di lavoro che concretizzano i principi della crittografia end-to-end (E2E) e del Proof-of-Work (PoW) eseguito lato client:

- La generazione sicura delle chiavi e la registrazione del "Creator";
- la creazione, crittografia, mining e firma di un nuovo blocco;
- il recupero e la decifratura sicura dei dati.

Elemento cardine di queste funzionalità è l'utility `frontend/src/utils/cryptoUtils.js`, che sfrutta la *Web Crypto API* del browser per eseguire operazioni crittografiche conformi agli standard internazionali.

8.2 L'Engine Crittografico Client-Side (cryptoUtils.js)

Prima di analizzare i componenti dell'interfaccia utente, è utile comprendere il funzionamento dell'engine crittografico che li sostiene. Il file `cryptoUtils.js` incapsula la complessità della Web Crypto API all'interno di un insieme di funzioni riutilizzabili, implementando la logica teorica descritta nei capitoli precedenti.

8.2.1 Generazione e Validazione delle Chiavi

Il sistema adotta la crittografia asimmetrica RSA per l'identità e per lo scambio sicuro delle chiavi. Le principali funzioni sono:

- `generateRSAKeyPair()`: Utilizza `crypto.subtle.generateKey` per creare, direttamente nel browser, una coppia di chiavi RSA-OAEP a 2048 bit.
- `exportKey()`: Converte le chiavi binarie (CryptoKey) nel formato standard PEM (Base64-encoded) che viene utilizzato per l'archiviazione (la pubblica) e per il download (la privata).
- `validatePrivateKeyPem()` e `verifyKeyPair()`: Funzioni di utilità fondamentali. Prima di eseguire qualsiasi operazione, il client verifica localmente che la chiave privata fornita dall'utente corrisponda alla chiave pubblica associata al suo `display_name` (recuperata dal server). Questo previene errori e tentativi di firma/decifratura con la chiave sbagliata.

8.2.2 Crittografia Ibrida (E2E)

Come descritto nell'architettura (Capitolo 5), la crittografia dei dati sensibili si basa su un approccio ibrido:

- `generateAESKey()` e `generateIV()`: Generano una chiave simmetrica AES-256-GCM (32 byte) e un vettore di inizializzazione (16 byte) effimeri per ogni singolo blocco.
- `encryptAESData()`: cifra i dati sensibili (`data_text`) utilizzando la chiave AES e l'IV. L'uso di GCM garantisce sia la confidenzialità che l'autenticazione dei dati cifrati.

- `encryptAESKeyWithPublicKey()`: Cifra la chiave AES effimera con la chiave pubblica RSA del "Creator" (recuperata dal server).

Il risultato di questo processo è un payload composto da `encrypted_data`, `data_iv` e `encrypted_data_key`: un insieme di dati completamente indecifrabili per il server.

8.2.3 Proof-of-Work e Firma Digitale

Le funzioni di integrità e "impegno computazionale" sono anch'esse gestite localmente.

- `buildHashInput()`: Una funzione deterministica che assembla tutti i componenti del blocco (hash precedente, hash dei dati cifrati, nonce, timestamp, ecc.) in una stringa standard.
- `hashData()`: Calcola l'hash SHA-256 della stringa di input.
- `mineBlock()`: Implementa il PoW client-side. È un ciclo `while` che incrementa un `nonce`, ricalcola l'hash fino a quando non soddisfi la difficoltà (vedi Listato 7.1).
- `signData()`: Una volta trovato l'hash valido, questa funzione lo firma utilizzando la chiave privata RSA del "Creator" per generare la firma digitale.

```
1  export const mineBlock = async (blockData, difficulty, timeoutMs)
       => {
2      let nonce = 0n; // Usa BigInt per nonce grandi
3      let hash = '';
4      const requiredPrefix = '0'.repeat(difficulty);
5      const startTime = Date.now();
6      const MAX_NONCE_CHECKS_BEFORE_YIELD = 10000;
7
8      console.log(`Starting client-side mining (difficulty: ${
           difficulty})...`);
9      try {
10         while (true) {
11             nonce++;
12             const currentBlockData = { ...blockData, nonce: nonce
                   };
13             // Costruisce la stringa da "hashare"
14             const hashInput = buildHashInput(currentBlockData);
15             // Calcola l'hash
16             hash = await hashData(hashInput);
17
18             // Controlla se l'hash soddisfa la difficolta
19             if (hash.startsWith(requiredPrefix)) {
20                 const duration = Date.now() - startTime;
```

```
21                  console.log(`Mining successful! Nonce: ${nonce},
                        Hash: ${hash...}, Duration: ${duration}ms`);
22                  return { nonce: nonce.toString(), hash, duration };
23              }
24
25              // Controllo timeout
26              if (Date.now() - startTime > timeoutMs) {
27                  console.warn(`Mining timeout exceeded after ${Date.
                        now() - startTime}ms`);
28                  throw new Error('Mining timeout exceeded');
29              }
30
31              // Permette al browser di "respirare" per non bloccare
                    l'UI
32              if (Number(nonce) % MAX_NONCE_CHECKS_BEFORE_YIELD ===
                    0) {
33                  await new Promise(resolve => setTimeout(resolve, 0)
                        );
34              }
35          }
36      } catch (error) {
37          console.error("Mining failed:", error);
38          throw error;
39      }
40  };
```

Listing 8.1: Implementazione del Proof-of-Work client-side in `cryptoUtils.js`.

8.3 Flusso 1: Registrazione di un Creator (CreatorRegistration.jsx)

Il primo componente che sfrutta `cryptoUtils.js` è la schermata di registrazione, progettata per garantire che la chiave privata non venga mai trasmessa al server.

1. L'utente inserisce un `display_name` (es. "Ospedale_Cardiologia").
2. Cliccando "Genera Chiavi", il componente chiama `generateRSAKeyPair()`.
3. Le chiavi vengono esportate in formato PEM tramite `exportKey()`.
4. La **chiave pubblica** (PEM) viene automaticamente inserita nel form e sarà inviata al backend.
5. La **chiave privata** (PEM) viene mostrata all'utente e resa disponibile per il download. Un avviso di sicurezza informa l'utente che questa chiave è l'unica responsabile della decifratura futura e deve essere salvata in un luogo sicuro.

6. Al submit, il componente invia solo il `display_name` e la `public_key_pem` all'endpoint `POST /creators` (analizzato nel Capitolo 6).

Questo flusso realizza il "permissioning" basato sull'identità crittografica, senza che il server gestisca mai alcuna informazione privata.

8.4 Flusso 2: Creazione e Mining del Blocco (BlockCreation.jsx)

Questo è il flusso più complesso e rappresenta il cuore dell'architettura ibrida. Segue la logica "a due fasi" introdotta nel Capitolo 6.

8.4.1 Fase 1: Preparazione e Validazione Chiave

L'utente compila il form inserendo il "Creator", i dati sensibili (`data_text`) e la propria chiave privata. Al click su "Avvia Creazione Blocco", il componente:

1. **Chiama l'API di Preparazione**: Esegue una `POST` a `/blocks/prepare-mining` inviando `display_name` e `data_text`.
2. **Riceve i Dati**: Il backend risponde con `creator_id`, `public_key_pem` (la chiave pubblica di quel creator), `previous_hash` e la `difficulty`.
3. **Verifica Corrispondenza Chiavi**: Il client esegue immediatamente `verifyKeyPair()` confrontando la `public_key_pem` ricevuta dal server con la `private_key_pem` inserita dall'utente. Se la verifica fallisce, il processo si interrompe.

8.4.2 Fase 2: Crittografia, Mining e Commit (Locali)

Se la verifica delle chiavi ha successo, il componente avvia il processo client-side (Listato 7.2), mostrando uno stato di "Mining locale".

```
1 // Questo useEffect viene triggerato dopo che la Fase 1 (prepare-
     mining)
2 // ha avuto successo e lo stato e' impostato su 'verifying'.
3 useEffect(() => {
4     const performClientSideOperations = async () => {
5         if (miningState.status === 'verifying' && preparationData
              && privateKeyPem) {
6
7             // 1. VERIFICA CHIAVI
8             toast.loading('Verifica corrispondenza chiavi...', { id
                  : 'key-verify' });
9             const isValidPair = await cryptoUtils.verifyKeyPair(
```

```
10                  preparationData.public_key_pem,
11                  privateKeyPem
12              );
13
14              if (!isValidPair) {
15                  toast.error('Chiave privata non corrisponde...', {
                        id: 'key-verify' });
16                  setMiningState({ status: 'failed', error: 'Key
                        mismatch' });
17                  return;
18              }
19              toast.success('Chiavi verificate!', { id: 'key-verify'
                    });
20              setMiningState({ status: 'mining', startTime: Date.now
                    () });
21
22              let encryptedData, iv, encryptedAesKey, dataSize;
23              try {
24                  // 2. CRITTOGRAFIA IBRIDA (E2E)
25                  const aesKey = cryptoUtils.generateAESKey();
26                  iv = cryptoUtils.generateIV();
27                  // Crittografa i dati con AES
28                  const ciphertext = await cryptoUtils.encryptAESData
                        (dataText, aesKey, iv);
29                  // Crittografa la chiave AES con RSA (chiave
                        pubblica del server)
30                  encryptedAesKey = await cryptoUtils.
                        encryptAESKeyWithPublicKey(
31                      aesKey,
32                      preparationData.public_key_pem
33                  );
34                  encryptedData = ciphertext;
35                  dataSize = encryptedData.byteLength + iv.byteLength
                        + encryptedAesKey.byteLength;
36              } catch (error) {
37                  toast.error(`Errore crittografia: ${error.message
                        }`);
38                  setMiningState({ status: 'failed', error: `
                        Encryption error: ${error.message}` });
39                  return;
40              }
41
42              try {
43                  const createdAt = new Date();
44                  const blockDataForMining = {
45                      previous_hash: preparationData.previous_hash,
46                      encrypted_data: encryptedData,
47                      data_iv: iv,
48                      encrypted_data_key: encryptedAesKey,
```

```
                    created_at: createdAt,
                    creator_id: preparationData.creator_id,
                    difficulty: preparationData.difficulty,
                };

                // 3. MINING PROOF-OF-WORK (Client-Side)
                const miningResult = await cryptoUtils.mineBlock(
                    blockDataForMining,
                    preparationData.difficulty,
                    CLIENT_SIDE_MINING_TIMEOUT_MS
                );

                // 4. FIRMA DIGITALE
                let signature;
                try {
                    // Firma l'hash risultante con la chiave privata
                    signature = await cryptoUtils.signData(privateKeyPem, miningResult.hash);
                } catch (error) {
                    toast.error(`Errore firma: ${error.message}`);
                    setMiningState({ status: 'failed', error: `Signing error: ${error.message}` });
                    return;
                }

                // 5. COMMIT DEL BLOCCO (Invio al Backend)
                const commitData = {
                    creator_id: preparationData.creator_id,
                    block_hash: miningResult.hash,
                    nonce: miningResult.nonce.toString(),
                    difficulty: preparationData.difficulty,
                    encrypted_data_hex: cryptoUtils.abToHex(encryptedData),
                    data_iv_hex: cryptoUtils.abToHex(iv),
                    encrypted_data_key_hex: cryptoUtils.abToHex(encryptedAesKey),
                    data_size: dataSize,
                    signature_hex: cryptoUtils.abToHex(signature),
                    created_at_iso: createdAt.toISOString(),
                    mining_duration_ms: miningResult.duration,
                };

                setMiningState({ status: 'committing' });
                commitBlockMutation.mutate(commitData); // Chiama l'API /blocks/commit

            } catch (error) { // Cattura errori di mining (es. timeout)
```

```
91                  toast.error(`Errore mining: ${error.message}`);
92                  setMiningState({ status: 'failed', error: `Mining
                        error: ${error.message}` });
93              }
94          }
95      };
96
97      performClientSideOperations();
98  }, [miningState.status, preparationData, privateKeyPem, dataText]);
```

Listing 8.2: Orchestrazione della crittografia e del mining in `BlockCreation.jsx`.

Il backend (come visto nel Capitolo 6) riceve questo blocco "pre-minato" e "pre-firmato", esegue le proprie validazioni (verifica della firma, verifica del PoW) e, se corretto, lo salva nel database.

8.5 Flusso 3: Decifratura dei Dati (DataDecryption.jsx)

Questo flusso conclude il ciclo E2E, dimostrando che soltanto il legittimo possessore della chiave privata può accedere ai dati in chiaro.

1. L'utente seleziona il proprio `display_name` e fornisce la propria `private_key_pem`.
2. **Verifica Chiave**: Come nel flusso di creazione, il componente chiama prima `/creators/.../public-key` e poi esegue `verifyKeyPair()`.
3. **Recupero Dati Cifrati**: Se la chiave è valida, il componente chiama l'endpoint `GET /decrypt/blocks/:creator_id`.
4. **Risposta del Server**: Il server risponde con un array di blocchi contenenti *solo* i dati crittografati (es. `encrypted_data_b64`, `data_iv_b64`, `encrypted_data_key_b64`).
5. **Decifratura Locale**: Il componente itera su ciascun blocco ricevuto, come mostrato nel Listato 7.3.

```
1  // Questo snippet e parte della funzione 'processDecryption'
2  // che viene eseguita dopo che le chiavi sono state verificate
3  // e i blocchi crittografati sono stati scaricati.
4
5  if (decryptionState.status === 'decrypting' && decryptionState.
       blocksToDecrypt && privateKeyPem) {
6      toast.loading(`Decifratura di ${decryptionState.blocksToDecrypt
           .length} blocchi...`, { id: 'decrypting-blocks' });
```

```
7
8       const blocks = decryptionState.blocksToDecrypt;
9       const results = [];
10
11      // Esegue la decifratura in "chunk" per non bloccare il browser
12      const chunkSize = 10;
13      for (let i = 0; i < blocks.length; i += chunkSize) {
14          const chunk = blocks.slice(i, i + chunkSize);
15
16          const chunkPromises = chunk.map(async (block) => {
17              try {
18                  // Converte da Base64 (JSON) ad ArrayBuffer
19                  const encryptedAesKey = cryptoUtils.base64ToAb(
                        block.encrypted_data_key_b64);
20
21                  // 1. DECIFRA CHIAVE AES (con Chiave Privata RSA)
22                  const aesKeyBytes = await cryptoUtils.
                        decryptAESKeyWithPrivateKey(
23                      encryptedAesKey,
24                      privateKeyPem
25                  );
26
27                  const iv = cryptoUtils.base64ToAb(block.data_iv_b64
                        );
28                  const ciphertext = cryptoUtils.base64ToAb(block.
                        encrypted_data_b64);
29
30                  // 2. DECIFRA I DATI (con Chiave AES recuperata)
31                  const plaintext = await cryptoUtils.decryptAESData(
32                      ciphertext,
33                      aesKeyBytes,
34                      iv
35                  );
36
37                  // Successo
38                  return { ...block, decrypted_data: plaintext, error
                        : null };
39
40              } catch (error) {
41                  // Fallimento (chiave errata, dati corrotti, ecc.)
42                  console.error(`Failed to decrypt block #${block.
                        block_number}:`, error);
43                  return { ...block, decrypted_data: null, error: '
                        Decryption failed', error_details: error.message
                         };
44              }
45          });
46
47          const chunkResults = await Promise.all(chunkPromises);
```

```
48          results.push(...chunkResults);
49
50          // Aggiorna l'UI e cede il controllo al browser
51          toast.loading(`Decifratura ${i + chunk.length}/${blocks.
                length} blocchi...`, { id: 'decrypting-blocks' });
52          await new Promise(resolve => setTimeout(resolve, 0));
53      }
54
55      //... (aggiorna lo stato con i risultati) ...
56      setDecryptedBlocksData({
57          //...
58          blocks: results,
59      });
60      setDecryptionState({ status: 'completed' });
61      toast.success(`Decifratura completata!`, { id: 'decrypting-
            blocks' });
62  }
```

Listing 8.3: Processo di decifratura client-side in `DataDecryption.jsx`.

Se la chiave privata è errata o i dati sono corrotti, la funzione `decryptAESData` (che usa AES-GCM) fallirà, generando un errore e dimostrando l'integrità del sistema.

8.6 Conclusione del Capitolo

L'implementazione del client rappresenta la realizzazione pratica dei principi di sicurezza e privacy alla base dell'architettura ibrida. Delegando al browser dell'utente la generazione delle chiavi, la crittografia end-to-end, la firma digitale e l'esecuzione del Proof-of-Work, il sistema garantisce che i dati sensibili e le chiavi private non siano mai esposti al server.

Il backend assume un ruolo notarile: si limita a convalidare e custodire blocchi cifrati, senza mai poterli decifrare. Il frontend, invece, diventa una vera e propria "cassaforte personale", l'unico ambiente in grado di creare, cifrare e successivamente sbloccare i dati.

Nei capitoli successivi verranno analizzate le prestazioni e le implicazioni di sicurezza di questo modello, evidenziando come la distribuzione del carico computazionale sul lato client contribuisca a rafforzare la resilienza e la trasparenza complessiva del sistema.

Capitolo 9

Architettura di Deployment e Orchestrazione

9.1 Introduzione

Dopo aver analizzato l'implementazione logica del backend (Capitolo 6) e del client (Capitolo 7), questo capitolo finale si concentra sull'architettura di deployment del sistema. Un obiettivo chiave del progetto è la creazione di una rete decentralizzata di nodi validatori. Per simulare questo ambiente in modo efficiente, riproducibile e isolato, è stata utilizzata la tecnologia di containerizzazione **Docker**.

L'intero sistema, composto da:

- Tre nodi validatori (backend Node.js)
- Tre database PostgreSQL distinti (uno per ogni nodo)
- Un'interfaccia client (frontend React)
- Un reverse proxy (Nginx) come gateway di accesso

È definito e orchestrato tramite un singolo file `docker-compose.yml`. Questo approccio garantisce la **portabilità** (il sistema può essere eseguito su qualsiasi macchina con Docker) e la **coerenza** (le dipendenze e le configurazioni di rete sono fisse), elementi cruciali per testare un'applicazione distribuita.

In questo capitolo analizzeremo i tre artefatti principali di questa architettura: i `Dockerfile` che definiscono le immagini, il file `docker-compose.yml` che orchestra i servizi e la configurazione `nginx.conf` che gestisce il traffico di rete.

9.2 Definizione delle Immagini (Dockerfile)

Un `Dockerfile` è un documento di testo che contiene tutte le istruzioni necessarie per assemblare un'immagine container. Il progetto ne utilizza due: uno per il backend e uno per il frontend.

9.2.1 Dockerfile del Backend (Nodo Validatore)

Il `Dockerfile` del backend (Listato 8.1) è responsabile della creazione di un'immagine leggera e ottimizzata per l'esecuzione del server Node.js.

```
1  # Usa un'immagine Node.js leggera basata su Alpine
2  FROM node:20-alpine
3
4  # Imposta la directory di lavoro
5  WORKDIR /app
6
7  # Copia i file package.json e package-lock.json
8  COPY package*.json ./
9
10 # Installazione delle dipendenze di produzione
11 RUN npm install --omit=dev
12
13 # Copia il resto del codice sorgente del backend
14 COPY . .
15
16 # Esponi le porte definite nel docker-compose.yml
17 EXPOSE 4001
18 EXPOSE 6001
19
20 # Comando per avviare l'applicazione
21 CMD ["npm", "start"]
```

Listing 9.1: Definizione dell'immagine del nodo validatore (`backend/Dockerfile`)

L'analisi del file mette in evidenza alcuni elementi chiave:

- `FROM node:20-alpine`: la scelta dell'immagine `alpine` è strategica, poiché riduce significativamente la dimensione finale del container e, di conseguenza, la superficie d'attacco.
- `RUN npm install --omit=dev`: vengono installate esclusivamente le dipendenze necessarie in produzione, escludendo i pacchetti di sviluppo (come `nodemon`), in linea con le best practice di sicurezza e performance.
- `EXPOSE 4001 6001`: vengono documentate le porte che il container renderà accessibili. Sarà poi `docker-compose` a occuparsi della loro mappatura. In

particolare, la porta 4001 gestisce le API REST, mentre la 6001 è dedicata al protocollo P2P basato su WebSocket.

Un `Dockerfile` analogo è utilizzato per il frontend, con la differenza che il comando finale (`CMD`) avvia il server di sviluppo Vite tramite `npm run dev`, in conformità con l'architettura di sviluppo del progetto.

9.3 Orchestrazione dei Servizi (docker-compose.yml)

Il file `docker-compose.yml` rappresenta il vero e proprio punto di orchestrazione dell'intero sistema, coordinando la definizione e la connessione di tutti i servizi. Il Listato 8.2 mostra un estratto significativo della configurazione.

```
services:
  # --- NGINX REVERSE PROXY & ENTRYPOINT ---
  nginx:
    image: nginx:stable-alpine
    container_name: blockchain-nginx
    ports:
      # Espone la porta HTTP standard per l'accesso esterno
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    networks:
      - blockchain-network
    depends_on:
      - node1
      - frontend

  # --- NODO 1 (API & P2P HUB) ---
  postgres1:
    image: postgres:16-alpine
    container_name: blockchain-postgres-1
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      # Volume per persistenza dati
      - postgres_data_1:/var/lib/postgresql/data
    networks:
      - blockchain-network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U $$POSTGRES_USER -d
          $$POSTGRES_DB"]
      interval: 5s
      timeout: 5s
```

```
34        retries: 5
35
36    node1:
37      build: ./backend
38      container_name: blockchain-node-1
39      environment:
40        - PORT=4001
41        - P2P_PORT=6001
42        # Connessione al proprio database
43        - DATABASE_URL=postgres://${POSTGRES_USER}:${
              POSTGRES_PASSWORD}@postgres1:5432/${POSTGRES_DB}
44        - PEERS=ws://node2:6002,ws://node3:6003
45        - DIFFICULTY=${DIFFICULTY:-4}
46      volumes:
47        - ./backend/src:/app/src
48      networks:
49        - blockchain-network
50      depends_on:
51        postgres1:
52          condition: service_healthy
53
54    # --- NODO 2 (node2) e NODO 3 (node3) ---
55    # (Configurazioni simili a node1 e postgres1 omessi)
56
57    # --- FRONTEND (Si collega a Nginx) ---
58    frontend:
59      build:
60        context: ./frontend
61      container_name: blockchain-frontend
62      volumes:
63        - ./frontend:/app
64        - /app/node_modules
65      ports:
66        - "5173:5173"
67      networks:
68        - blockchain-network
69      depends_on:
70        - node1
71
72  volumes:
73    postgres_data_1:
74    postgres_data_2:
75    postgres_data_3:
76
77  networks:
78    blockchain-network:
79      driver: bridge
```

Listing 9.2: Estratto dell'orchestrazione dei servizi (`docker-compose.yml`)

Questa configurazione realizza la simulazione di una rete decentralizzata composta da più nodi. Gli aspetti principali che emergono sono:

- **Servizi distinti**: ogni nodo (`node1`, `node2`, `node3`) è definito come servizio indipendente, ma costruito a partire dallo stesso `Dockerfile` del backend, garantendo uniformità e modularità
- **Persistenza Isolata**: Ogni nodo ha il proprio database PostgreSQL (`postgres1`, `postgres2`, `postgres3`). Le direttive `volumes` (ad esempio `postgres_data_1`) assicurano la persistenza dei dati sul disco dell'host, consentendo la conservazione dello stato anche dopo un riavvio dei container.
- **Rete privata**: tutti i servizi sono connessi a una rete Docker interna (`blockchain-network`), che garantisce un ambiente isolato e sicuro per la comunicazione tra container.
- **Service discovery automatica**: Docker Compose gestisce internamente la risoluzione DNS, permettendo ai container di comunicare tramite il nome del servizio (ad esempio, `node1` si connette a `postgres1` attraverso `DATABASE_URL`).
- **Simulazione P2P**: La rete P2P è creata passando le variabili d'ambiente. `node1` riceve `PEERS=ws://node2:6002,ws://node3:6003`. Grazie al service discovery, `node1` può contattare `node2` e `node3` usando i loro nomi di servizio.
- **Ordine di avvio controllato**: la direttiva `depends_on` e l'uso dell'opzione `healthcheck` su PostgreSQL assicurano che ogni nodo del backend venga avviato solo dopo che il rispettivo database è operativo.

9.4 Il Gateway di Rete (nginx.conf)

Nessuno dei servizi backend o frontend è esposto direttamente al mondo esterno, ad eccezione del servizio `nginx`. Nginx agisce come unico punto di accesso (*entrypoint*), un reverse proxy che instrada il traffico ai servizi interni corretti.

```
1  http {
2      # Definizione dei server "upstream" (i nostri servizi interni)
3      upstream backend_api {
4          server node1:4001;
5          # In futuro, si potrebbe bilanciare il carico su tutti i
               nodi:
6          # server node2:4002;
7          # server node3:4003;
8      }
9
10     upstream frontend {
11         server frontend:5173;
```

```
12      }
13
14      # Server principale che ascolta sulla porta 80
15      server {
16          listen 80;
17          server_name localhost;
18
19          # Rotta per l'API
20          location /api/ {
21              # Rimuove /api/ dal percorso prima di inoltrarlo
22              rewrite ^/api/(.*)$ /$1 break;
23
24              proxy_pass http://backend_api;
25
26              # Intestazioni per inoltrare correttamente le
                    informazioni del client
27              proxy_set_header Host $host;
28              proxy_set_header X-Real-IP $remote_addr;
29              proxy_set_header X-Forwarded-For
                    $proxy_add_x_forwarded_for;
30              proxy_set_header X-Forwarded-Proto $scheme;
31          }
32
33          # Rotta principale (catch-all) per il frontend
34          location / {
35              proxy_pass http://frontend;
36
37              # Intestazioni necessarie per il server di sviluppo
                    Vite (WebSocket)
38              proxy_http_version 1.1;
39              proxy_set_header Upgrade $http_upgrade;
40              proxy_set_header Connection 'upgrade';
41              proxy_set_header Host $host;
42          }
43      }
44  }
```

Listing 9.3: Estratto della configurazione del reverse proxy (`nginx/nginx.conf`)

La configurazione di Nginx è fondamentale per due motivi:

1. **Unificazione dell'Interfaccia**: Per l'utente finale (e per il client React), l'intero sistema è accessibile su un'unica porta (`80`). Il client non ha bisogno di sapere che l'API si trova sulla porta `4001` e il frontend sulla `5173`.

2. **Routing basato su Prefisso**: La direttiva `location` è il cuore di questa logica.

- `location /api/ { ... }`: Qualsiasi richiesta che inizia con `/api/` viene inoltrata al `backend_api` (definito come `node1:4001`). La direttiva `rewrite` rimuove il prefisso `/api/`, in modo che il backend riceva una richiesta pulita (es. `/creators` invece di `/api/creators`).
- `location / { ... }`: Qualsiasi altra richiesta viene inoltrata al servizio `frontend`, permettendo il caricamento dell'applicazione React.

9.5 Conclusione del Capitolo

L'integrazione di Docker, Docker Compose e Nginx consente di ottenere un'architettura di deployment robusta, portabile e adatta alla simulazione di una rete distribuita multi-nodo.

I `Dockerfile` garantiscono la riproducibilità dell'ambiente di esecuzione, mentre `docker-compose.yml` si occupa dell'orchestrazione dei servizi, della gestione delle reti e della persistenza dei dati per tutti gli otto componenti principali (tre nodi, tre database, un frontend e un proxy). Infine, Nginx svolge il ruolo di "controllore del traffico", fornendo un punto di accesso unico e sicuro che nasconde la complessità della rete interna e consente un'interazione trasparente con l'applicazione distribuita.

Capitolo 10

Showcase Applicativo e Interfaccia Utente

10.1 Introduzione

Dopo aver analizzato in dettaglio l'architettura teorica (Capitolo 5), l'implementazione del nodo validatore (Capitolo 6), l'infrastruttura di deployment (Capitolo 8) e il "client pesante" crittografico (Capitolo 7), questo capitolo presenta l'interfaccia utente (UI) finale del sistema.

L'applicazione frontend, sviluppata in React, funge da punto di accesso unificato per tutte le funzionalità del sistema. Come discusso in precedenza, questa interfaccia non è un semplice livello di presentazione, ma è la componente attiva responsabile dell'esecuzione di tutte le operazioni crittografiche sensibili: dalla generazione delle chiavi alla crittografia end-to-end, fino al mining del Proof-of-Work e alla firma digitale.

Nelle sezioni seguenti, verranno mostrate le cinque schermate principali che compongono l'applicazione. Ogni schermata corrisponde a un flusso logico fondamentale del sistema e dimostra visivamente l'interazione dell'utente con la blockchain e con i dati protetti. Le immagini sono state catturate su un'istanza del sistema preventivamente inizializzata con dati di esempio.

10.2 Dashboard Principale

La Figura 10.1 mostra la schermata iniziale dell'applicazione. Questa dashboard fornisce una panoramica aggregata dello stato della rete, mostrando statistiche chiave in tempo reale recuperate dai nodi validatori. Il tempo di mining medio è calcolato automaticamente e la difficoltà impostata di default è di 4 zeri. Se venisse rappresentata la distribuzione statistica dei valori di hash, potremmo notare che questa sarebbe *uniforme*. Ciò significa che ogni possibile hash (un numero specifico tra 0 e 2^{256}) ha la stessa identica probabilità di uscire.

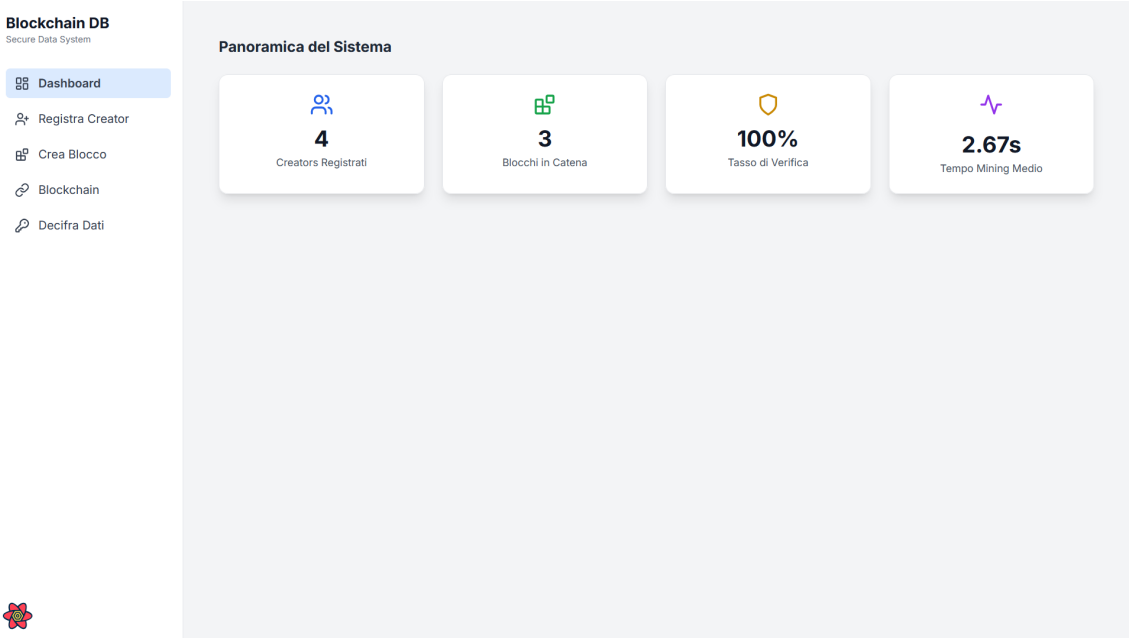

Figura 10.1: La dashboard principale del sistema.

10.3 Registrazione di un Creator

La Figura 10.2 illustra il processo di "permissioning", ovvero la registrazione di una nuova entità autorizzata a scrivere sulla blockchain. Come discusso nel Capitolo 7, la generazione delle chiavi RSA avviene interamente localmente. Dalla figura si può notare che la chiave pubblica è mostrata in chiaro, mentre la chiave privata viene "nascosta". Il bottone rappresentante l'occhio permette all'utente di visualizzare la chiave privata, mentre il bottone alla sua destra permette di copiarla automaticamente negli appunti. Sotto la chiave privata si può notare il bottone *Scarica Chiave Privata*, che permette di salvare nella cartella Download un file di testo contenente la chiave privata. Se quest'ultima non viene salvata/scaricata, l'utente rimarrà registrato nella catena del *Creators*, ma non sarà possibile creare blocchi nella catena (e di conseguenza decifrarli). La registrazione effettiva dell'utente avviene premendo il tasto *Registra Creator*.

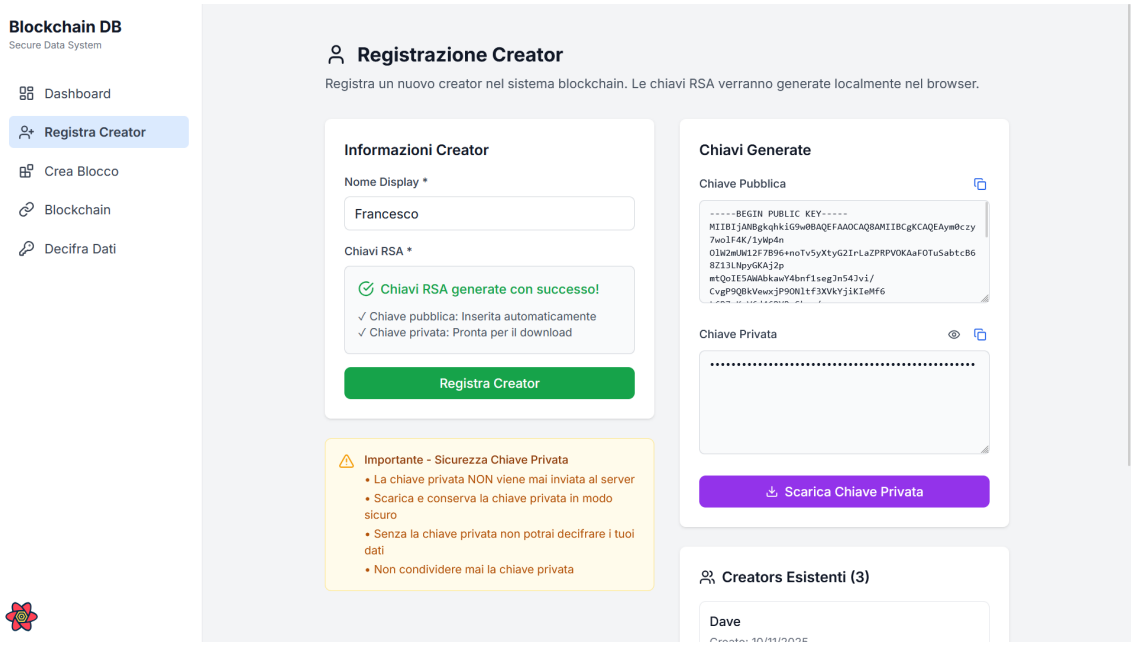

Figura 10.2: Interfaccia di generazione chiavi e registrazione Creator.

10.4 Creazione e Mining di un Blocco

La Figura 10.3 mostra il cuore dell'applicazione: il pannello di creazione di un nuovo blocco. Questa interfaccia implementa il flusso "a due fasi" discusso nel Capitolo 7. Come mostrato nella casella a destra della creazione, si può notare che è stato creato con successo un blocco relativo al *creator* Bob. Inoltre, sono mostrati altri dati di interesse come l'ID del blocco, l'hash del blocco, il nonce o il tempo di mining. Al di sotto della casella contenente la chiave privata dell'utente, è presente il bottone che permette l'aggiunta del blocco alla catena avviando il processo di mining.

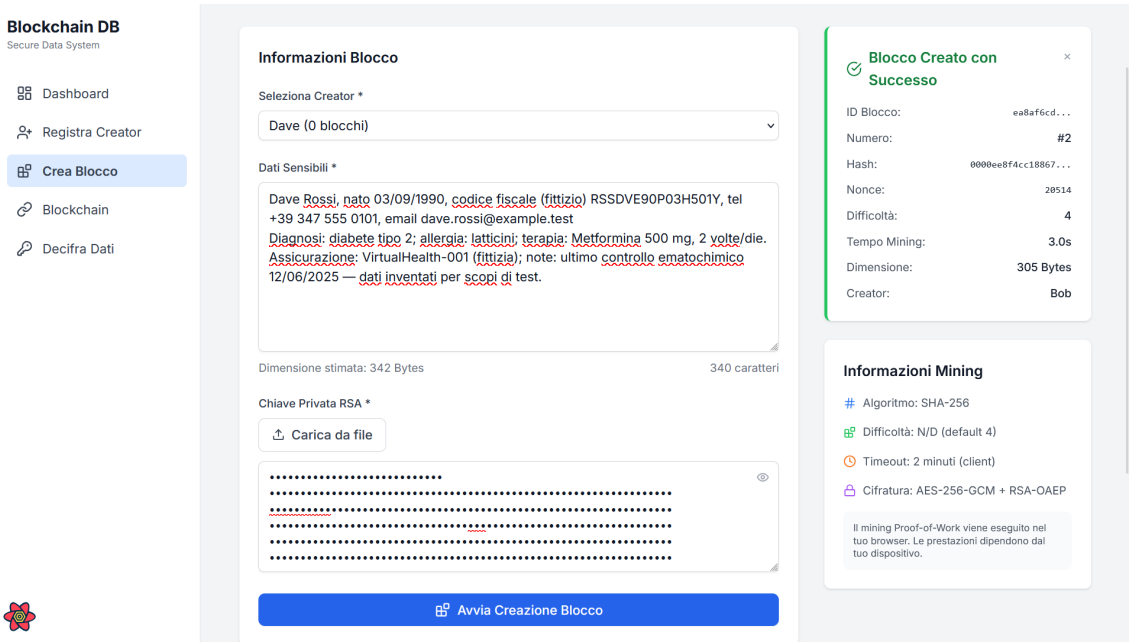

Figura 10.3: Pannello di creazione, crittografia e mining client-side.

10.5 Esplorazione della Blockchain

La Figura 10.4 presenta l'esploratore della catena. Questa vista permette a qualsiasi utente di ispezionare i metadati di tutti i blocchi presenti sulla catena, garantendo trasparenza sulle operazioni. Si può notare che non tutti i blocchi sono stati ancora validati, infatti il blocco numero 3 è in attesa di essere validato da tutti i nodi.

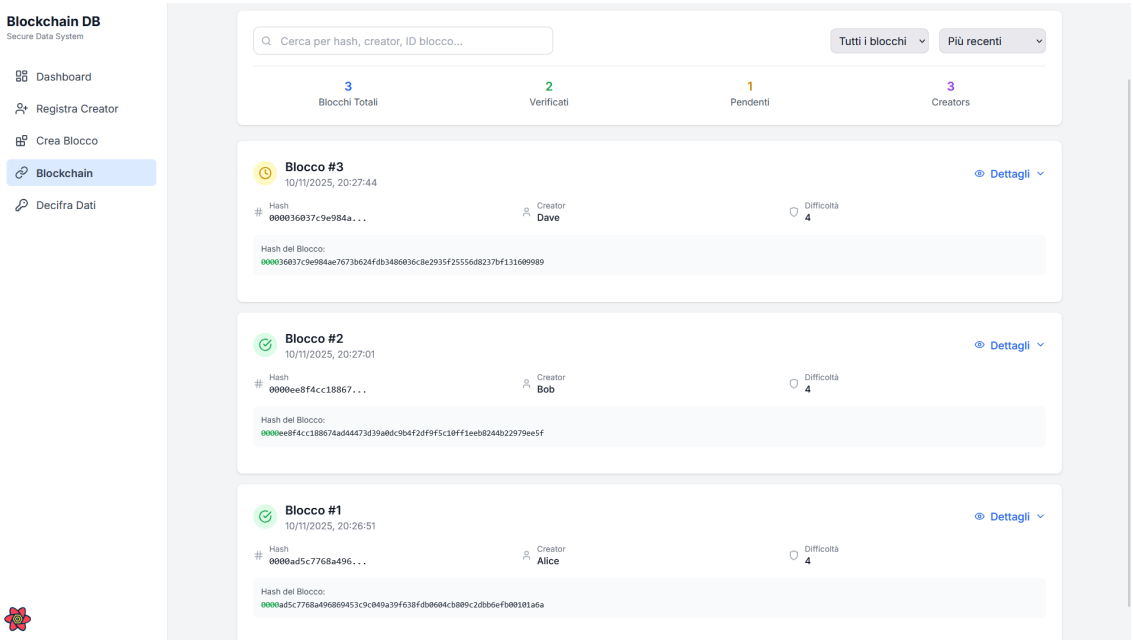

Figura 10.4: Interfaccia di esplorazione dei blocchi della catena.

10.6 Decifratura dei Dati Sensibili

Infine, la Figura 10.5 mostra il processo di decifratura end-to-end. Questa è l'operazione speculare a quella di creazione. Inserendo il nome del creator e caricando la sua chiave privata, verranno decifrati tutti i blocchi appartenenti all'utente. Come specificato nei capitoli precedenti, la decifratura avviene interamente lato client. In questo modo si può evitare un trasferimento della chiave privata in chiaro, che rappresenterebbe una vulnerabilità inaccettabile.
Il tasto "Scarica Tutti i dati Decifrati" permette il download formattato del contenuto di tutti i blocchi decifrati.

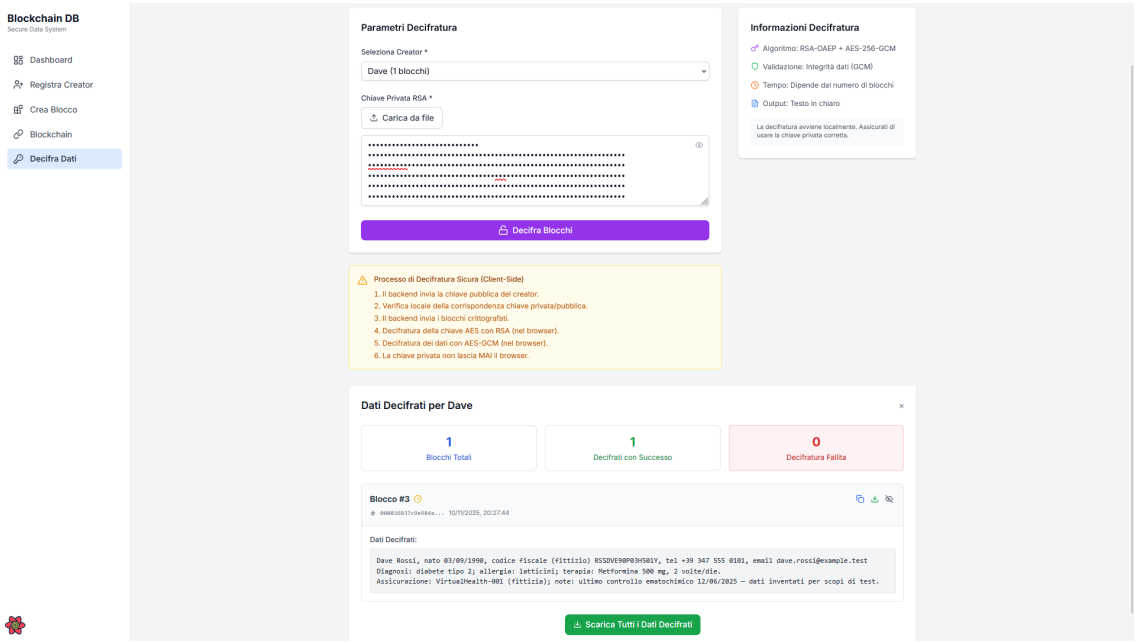

Figura 10.5: Pannello di decifratura client-side dei dati.

Capitolo 11

Conclusioni e Sviluppi Futuri

11.1 Sintesi del Progetto

Questo lavoro di tesi propone un'alternativa per l'archiviazione dei dati sensibili, unendo la maturità dei database relazionali con le garanzie di sicurezza della tecnologia Blockchain. Questo sistema ibrido mira a superare le criticità dei modelli centralizzati, in particolare il *single point of trust* (l'amministratore) e il *single point of failure* (il server), vulnerabili a manipolazioni interne e attacchi mirati.

La tecnologia Blockchain offre un'architettura fondata sulla decentralizzazione, l'immutabilità crittografica e la trasparenza, eliminando la necessità di fiducia che, nei database centralizzati, è riposta nell'ente intermediario. Tuttavia, le blockchain pubbliche classiche sono inadatte alla gestione dei dati sensibili a causa della loro totale trasparenza e delle loro scarse prestazioni nelle interrogazioni complesse.

L'obiettivo è stato quello di creare un sistema in cui il database relazionale funga da "livello di persistenza" efficiente, mentre una catena di blocchi (anch'essa memorizzata nel DB) agisca come un "registro notarile" immutabile che certifichi ogni singola operazione sui dati.

L'analisi architetturale e l'implementazione pratica (discusse nei capitoli precedenti) hanno dimostrato un modello funzionante in cui i dati sensibili e le **chiavi** private non sono mai **trasferite** in chiaro al server. Il trattamento **crittografico** (cifratura e decifratura) avviene interamente lato client, tramite una rigorosa crittografia end-to-end (E2E).

11.2 Punti di Forza dell'Architettura proposta

Il prototipo realizzato ha validato l'efficacia dell'architettura ibrida, raggiungendo i seguenti obiettivi chiave:

- **Immutabilità e Non Ripudio**: L'uso di una catena di blocchi concatenati crittograficamente, in cui ogni blocco salva l'impronta digitale (ovvero l'hash) del blocco precedente, garantisce l'**immutabilità** dei dati. Qualsiasi tentativo di alterare un record storico (un blocco) invaliderebbe l'intera catena successiva. Inoltre ogni blocco è firmato digitalmente (con la chiave privata RSA del "Creator") prima di essere inviato al server. Questo fornisce una forte garanzia di **non ripudio**: è possibile provare crittograficamente chi ha creato un dato e che tale dato non è stato alterato dopo la firma.

- **Privacy by Design (Crittografia E2E)**: Come dimostrato nei Capitoli 6 e 7, i dati sensibili vengono crittografati sul client (con una chiave AES effimera) prima di qualsiasi trasmissione. La chiave AES stessa viene crittografata con la chiave pubblica RSA del "Creator". Il server (backend e database) memorizza unicamente il *ciphertext* (`encrypted_data`, `data_iv`, `encrypted_data_key`). Di conseguenza, il sistema è *zero-knowledge* dal punto di vista del server: **neanche** un amministratore di database con pieni privilegi può accedere al contenuto dei dati in chiaro.

- **Separazione tra Dati e "Prova"**: Questa implementazione mostra i vantaggi del modello relazionale, **in cui** i metadati dei blocchi (timestamp, hash, creator_id) sono memorizzati in colonne SQL indicizzate, **permettendo** interrogazioni efficienti. I dati sensibili, **invece**, sono trattati come "payload" binari (`BYTEA`), separando l'efficienza di ricerca dalla sicurezza dei contenuti.

- **Difesa dal Tampering (Livello DB)**: Oltre alla garanzia crittografica, il sistema implementa una difesa a livello di database tramite un *trigger* PostgreSQL (Capitolo 6). Questo trigger rende la tabella `blockchain.blocks` *append-only* e impedisce qualsiasi tentativo di `DELETE` o `UPDATE` (eccetto per il flag di verifica).

- **Proof-of-Work come "Rate-Limiting"**: Il progetto reinterpreta in modo innovativo il Proof-of-Work. Non essendo utilizzato per il consenso (come in Bitcoin), il PoW viene eseguito sul client e funge da meccanismo anti-spam e "Proof of Commitment". Obbliga il client a spendere risorse computazionali per creare un blocco, impedendo attacchi di tipo *denial-of-service* volti a inondare i nodi validatori di blocchi spazzatura.

11.3 Limiti e Sviluppi Futuri

Nonostante il prototipo abbia raggiunto gli obiettivi prefissati, l'analisi ha evidenziato diversi limiti intrinseci, che aprono la strada a numerosi sviluppi futuri:

- **Gestione delle Chiavi**: La sicurezza dell'intero sistema E2E dipende dalla capacità dell'utente di custodire responsabilmente la propria chiave privata. L'attuale implementazione, che richiede di incollare o caricare la chiave privata per minare o decifrare blocchi, è funzionale ma fragile. Uno sviluppo futuro cruciale sarebbe l'integrazione con sistemi di gestione delle chiavi più robusti, come *Hardware Security Modules* (HSM), *Web Crypto API* (per la memorizzazione sicura nel browser) o sistemi di *Identità Decentralizzata* (DID).
- **Mancanza di Interrogazione sui Dati Cifrati**: Il vantaggio della privacy E2E introduce un limite non trascurabile: è impossibile per il server eseguire interrogazioni sul contenuto dei dati sensibili (es. `SELECT * WHERE dati_sensibili.nome = 'Alice'`). Per abilitare questa funzionalità, la ricerca futura dovrebbe esplorare tecniche di ***Searchable*** *Encryption* (SE) o, per operazioni più complesse, la *Crittografia Omomorfica* (HE).
- **Modello di Consenso Semplificato**: L'architettura di deployment (Capitolo 8) simula una rete P2P in cui i nodi si fidano e si aggiornano a vicenda tramite broadcast (`p2p.js`). Questo modello è collaborativo, ma non resistente a nodi maliziosi (bizantini). Uno sviluppo futuro dovrebbe implementare un vero algoritmo di consenso per reti *permissioned* (ad esempio *Proof of Authority*) per garantire la coerenza tra i nodi validatori.
- **Conformità GDPR (Diritto all'Oblio)**: L'immutabilità è un fondamento della tecnologia blockchain, ma è in conflitto con il "diritto all'oblio" sancito dal GDPR. Le evoluzioni future potrebbero esplorare tecniche di *cryptographic shredding*: **distruggendo** in modo sicuro e verificabile la chiave AES utilizzata per crittografare i dati, rendendo il *ciphertext* sulla catena permanentemente illeggibile.
- **Scalabilità**: Il modello "un blocco per un dato" è stato scelto per la sua semplicità, ma questo approccio non è scalabile per sistemi ad alta produttività (es. IoT, log). L'implementazione di un *Merkle Tree* (discusso nel Capitolo 4) per raggruppare migliaia di transazioni in un unico blocco potrebbe ridurre l'overhead sulla catena.

In conclusione, questo lavoro di tesi ha dimostrato con successo la fattibilità di un sistema ibrido che sposa l'efficienza dei database relazionali con la sicurezza della blockchain. Il prototipo implementato fornisce una solida base architetturale

per la gestione di dati sensibili, garantendo privacy e immutabilità attraverso una rigorosa separazione dei compiti tra un "client crittografico" e un "nodo validatore zero-knowledge".

Bibliografia

[1] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Permissioned blockchains: Properties, techniques and applications. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2813–2820, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457539. URL https://doi.org/10.1145/3448016.3457539.

[2] Adam Back et al. Hashcash-a denial of service counter-measure. 2002.

[3] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US. ISBN 978-1-4757-0602-4.

[4] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL https://doi.org/10.1145/362384.362685.

[5] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 37–54, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31870-5.

[6] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. Hybrid blockchain database systems: design and performance. *Proc. VLDB Endow.*, 15(5):1092–1104, January 2022. ISSN 2150-8097. doi: 10.14778/3510397.3510406. URL https://doi.org/10.14778/3510397.3510406.

[7] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 437–455, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-38424-3.

[8] Saba Khanum and Khurram Mustafa. A systematic literature review on sensitive data protection in blockchain applications. *Concurrency and Computa-*

tion: Practice and Experience, 35(1):e7422, 2023. doi: https://doi.org/10.1002/cpe.7422. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.7422.

[9] Jie Lian, Siqian Wang, and Yanmiao Xie. Tdrb: An efficient tamper-proof detection middleware for relational database based on blockchain technology. *IEEE Access*, 9:66707–66722, 2021. doi: 10.1109/ACCESS.2021.3076235.

[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009. URL http://www.bitcoin.org/bitcoin.pdf.

[11] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database, 2019. URL https://arxiv.org/abs/1903.01919.

[12] Shamkant B. Navathe Ramez Elmasri. *Fundamentals of Database Systems*. Pearson, 2010.

[13] Gautami Tripathi, Mohd Abdul Ahad, and Gabriella Casalino. A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges. *Decision Analytics Journal*, 9:100344, 2023. ISSN 2772-6622. doi: https://doi.org/10.1016/j.dajour.2023.100344. URL https://www.sciencedirect.com/science/article/pii/S2772662223001844.

[14] Hoang Tam Vo, Ashish Kundu, and Mukesh K. Mohania. Research directions in blockchain data management and analytics. In *International Conference on Extending Database Technology*, 2018. URL https://api.semanticscholar.org/CorpusID:3895194.