Politecnico
di Torino

Tesi di Laurea Magistrale in Ingegneria Informatica (Computer Engineering)

Event sourcing pattern applicato all'auditing e al tracciamento di modifiche ai contenuti

di

Enrico D'Oro

Relatore:
Prof. M. Torchiano

Tutor aziendale:
Alessandro Vidotto

Politecnico di Torino
2022

Abstract

L'*event sourcing pattern* è un approccio alternativo all'interazione con un *information system* rispetto al classico approccio CRUD (*Create*, *Read*, *Update* e *Delete*). Con l'*event sourcing*, infatti, i dati vengono memorizzati come una serie di eventi, conservando inoltre informazioni sul contesto di quest'ultimi. Proprio questo meccanismo di aggiunte singole di eventi immutabili a un database, rendono il pattern perfetto per fornire un servizio di *auditing* e non solo.

Se usato in combinazione con altri pattern, come il CQRS (*Command Query Responsibility Segregation*), può amplificare il tipo di funzionalità offerte, ponendo riparo ad alcuni svantaggi che si avrebbero se usato singolarmente. In particolare, questa combinazione di *pattern* permette di creare degli aggregati, utilizzabili come dato validato e pronto all'uso.

L'azienda Coolshop ha voluto utilizzare l'*event sourcing pattern* in un suo prodotto, denominato CoolPIM (la versione proprietaria del più generico applicativo *Product Information Management*), per implementare queste due funzionalità, a vantaggio delle soluzioni più classiche basate sull'approccio CRUD.

L'obiettivo di questa tesi è dunque quello di introdurre queste funzionalità e la loro realizzazione, dimostrandone il corretto funzionamento e l'effettivo vantaggio portato dal loro utilizzo per mezzo di test e studi di performance effettuati con *framework* appositi.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Con la mia esperienza presso l'azienda italiana Coolshop Srl, con sede a Torino, ho avuto modo di interfacciarmi col mondo dello sviluppo di applicazioni web e software ad hoc per clienti terzi, e di contribuire fin da subito alla crescita del loro applicativo CoolPIM, aggiungendo nuove funzionalità e senza impattare su quelle già esistenti.

1.1 L'azienda

Coolshop è un'azienda fornitrice di soluzioni digitali, centrate sull'esperienza utente e focalizzate sulla creazione di processi di vendita estesa. Fondata nel 2010 a Torino, conta attualmente ulteriori tre sedi nel resto del mondo: Rotterdam, Chicago e Dubai.

Oltre a seguire diversi progetti con più aziende, anche di settori differenti, fornisce principalmente prodotti per due tipi di soluzioni: Sales e PIM [2].

1.2 Le nuove funzionalità per il progetto CoolPIM

Il mio contributo all'azienda è stato volto all'introduzione di nuove funzionalità e alla modifica di alcune già presenti per il progetto CoolPIM, un'applicazione prodotta dall'azienda a partire da un generico software PIM, cui sono stati aggiunti servizi specifici richiesti dal cliente.

Queste nuove funzionalità sono entrambe basate sull'*event sourcing pattern*, la cui introduzione nel progetto ha gettato le basi per poter fornire eventualmente ulteriori nuovi servizi che sfruttino questo *pattern*, allargando le potenzialità dell'applicazione e fornendo funzionalità aggiuntive agli utenti finali.

La versione originale del CoolPIM era stata realizzata sfruttando il solo *event sourcing* che si è successivamente rivelato inadatto per offrire alcune tipologie di servizio: con l'aumentare del numero di eventi da memorizzare e dovendo soddisfare le richieste di più utenti contemporaneamente, questa soluzione completamente *event sourced* infatti si è rivelata essere inefficace nei tempi di risposta per fornire i dati richiesti e nell'affidabilità dei dati forniti, non sempre aggiornati con le ultime modifiche per una mancata sincronizzazione tra le varie operazioni di tutti gli utenti. Per questi motivi, l'azienda ha ritenuto opportuno adottare una soluzione più classica, basata su un approccio CRUD implementato sfruttando un database non relazionale (noSQL in questo caso specifico). Ciò nonostante, l'*event sourcing pattern* non è stato completamente abbandonato: è stato scelto infatti di prenderlo in considerazione per implementare queste nuove funzionalità, sfruttandone i vantaggi offerti come spiegato nei successivi capitoli.

Capitolo 2

Il progetto CoolPIM

Il CoolPIM è la versione fornita da Coolshop di un applicativo PIM (*Product Information Management*), ossia un *content management system*, utilizzato per raccogliere, gestire e arricchire in maniera coerente le informazioni di vari prodotti e inviarle a diversi canali di distribuzione.

Viene utilizzato in contesti dove serve mantenere uniformità e coerenza di dati in diversi punti della rete: in ambito di *e-commerce*, ad esempio, occorre mantenere le informazioni dei vari prodotti identiche nei vari punti vendita.

Un PIM è dunque una piattaforma collaborativa, che facilita le operazioni e la collaborazione fra i vari reparti aziendali, centralizzando i dati in un unico sistema e ridistribuendoli sui canali desiderati.

CoolPIM, in particolare, è progettata per gestire i dati di prodotti, dati multimediali e contenuti per arricchire l'esperienza di configurazione e l'offerta di cataloghi. Inoltre, consente di archiviare e organizzare il contenuto in modo da essere recuperato da sistemi esterni per creare layout di stampa avanzati e offrire diversi modelli [2].

2.1 La struttura del CoolPIM

Nella documentazione ufficiale fornita dall'azienda [3], viene identificata come entità principale del CoolPIM un albero orientato ai nodi foglia, con una relazione padre-figlio, ossia una struttura gerarchica, come illustrato in figura 2.1. Ogni

nodo rappresenta un prodotto con determinati attributi e può essere legato ad altri prodotti da una parentela che consentirà di ereditare (o tramandare) questi attributi ed eventualmente modificarli.

Questa struttura consente:

- di visualizzare complesse rappresentazioni della struttura dati, dove a svolgere un ruolo cruciale sono i nodi foglia e la loro relazione reciproca
- di cercare all'interno dei percorsi che definiscono le relazioni tra i nodi foglia

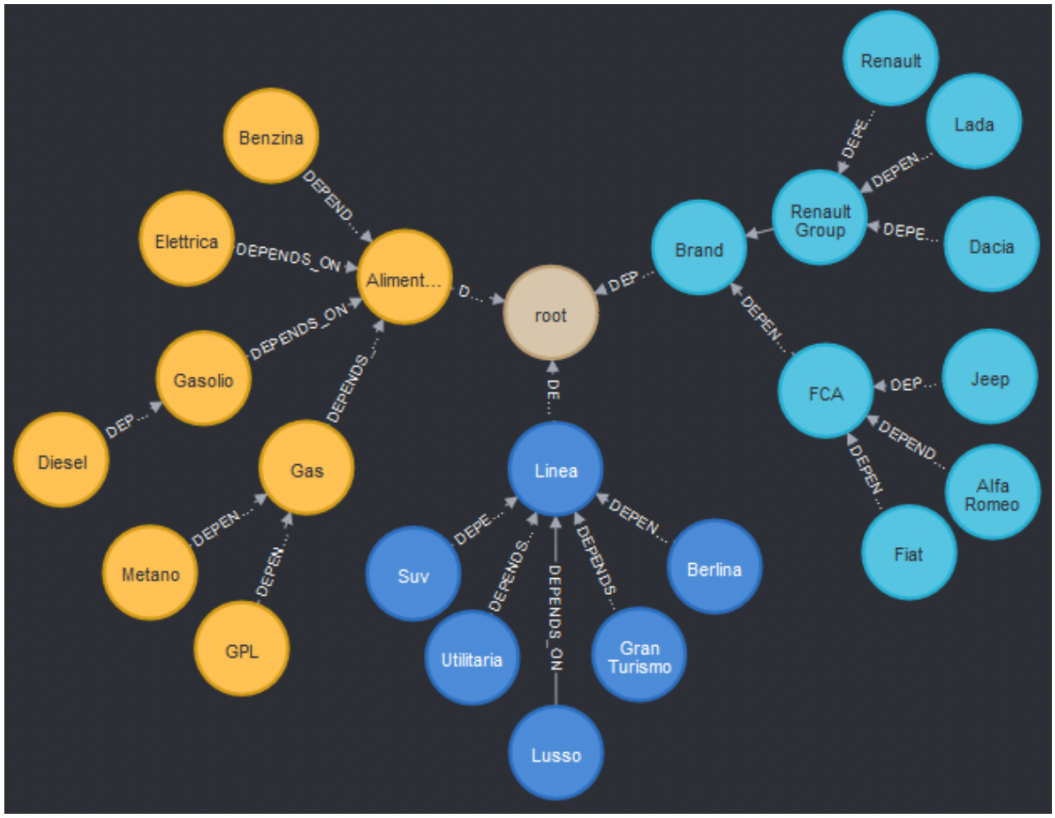


Figura 2.1 Esempio di struttura ad albero di prodotti.

Le relazioni dei nodi all'interno dell'albero sono gestite nell'interfaccia utente e memorizzate nel database. La relazione all'interno degli alberi è del tipo *many to many* e può essere raggiunta collegando nodi di diversi alberi (gerarchie).

2.1.1 Risoluzione delle dipendenze nell'entità nodo

I nodi vengono validati durante la creazione del nodo stesso, controllando che non vengano create delle dipendenze cicliche. Il sistema controlla dunque che la gerarchia sia costruita come in figura 2.2.

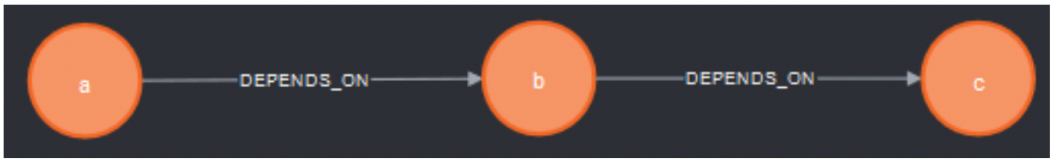


Figura 2.2 Esempio di dipendenza tra nodi.

La query utilizzata quando viene creato un nuovo nodo B come figlio di un altro nodo A, è costruita come segue:

- Cerca i nodi A e B (con A diverso da B e A tale per cui non possa creare una relazione con se stesso)
- Controlla il percorso che lega B ad A
- Se e solo se non sono stati trovati altri percorsi che collegano B ad A, viene creata una relazione tra A e B
- Infine, ritorna le proprietà della relazione appena creata (o `null` in caso contrario

2.1.2 Le entità principali

I nodi

Ogni nodo foglia è definito da un ID (generato come un GUID, ossia un identificatore unico globale), da una relazione padre-figlio (ossia l'ID del nodo padre) e da una collezione di attributi.

Attributi o *Content Type*

Gli attributi sono dati che permettono di descrivere il prodotto nel dettaglio. Gli attributi hanno un tipo che può essere: un testo, un'area di testo, un numero, una

misura, un'immagine, un file, una data, un *array* o una *select*. Un attributo può quindi definire una proprietà, basata sul suo tipo.

Gli attributi devono essere raggruppati in diverse collezioni chiamate *Content Type Group*, per ognuno dei quali deve essere definito almeno un attributo.

Con questa struttura è possibile definire una vasta gamma di contenuti, rendendoli scalabili anche grazie alla relazione gerarchica tra i nodi che fa sì che questi attributi possano essere ereditati e automaticamente pre-compilati.

La principale interazione tra un utente e il CoolPIM avviene attraverso queste entità e per questo motivo è stato deciso di registrare gli eventi scatenati da queste interazioni in un *event store* fornendo un servizio di *auditing* e non solo.

2.2 L'interfaccia utente

L'interfaccia utente del CoolPIM comprende quattro sezioni: *Dashboard*, *Language*, *Product Info* e *Hierarchies*.

Ognuna di queste sezioni richiede un tipo di interazione diversa da parte dell'utente che opererà soprattutto nella sezione *Hierarchies* una volta definiti i tipi di prodotto. Proprio da questa sezione vengono registrati gli eventi, resi poi disponibili come cronologia di modifiche in questa stessa sezione, e per creare degli aggregati con cui calcolare delle statistiche visualizzabili nella sezione *Dashboard*.

2.2.1 La sezione *Language*

Nella sezione *Language* è possibile visualizzare la lista di lingue disponibili o aggiungerne di nuove. Selezionando una lingua di default, sarà possibile visualizzare nelle altre sezioni le informazioni dei prodotti nella lingua specificata, purché siano state inserite al momento della creazione e definizione di un prodotto.

2.2.2 La sezione *Product Info*

L'accesso alla sezione *Product Info* è limitata a seconda dei permessi utente e permette di visualizzare tutte le strutture a nodi già disponibili, eliminarle o crearne

di nuove. Alla definizione di un prodotto segue la definizione dei suoi *Content Type Group*.

Un *Content Type Group* è un insieme di diversi attributi di un nodo, raggruppabili per natura o tipologia (ad esempio, un *Content Type Group* Dimensioni, che contiene i *Content Type* altezza, larghezza e lunghezza).

Un *Content Type* è dunque un attributo specifico di un nodo cui è associata una tipologia di dato specifica (ad esempio, l'attributo altezza è una tipologia di misura, con unità di misura il metro).

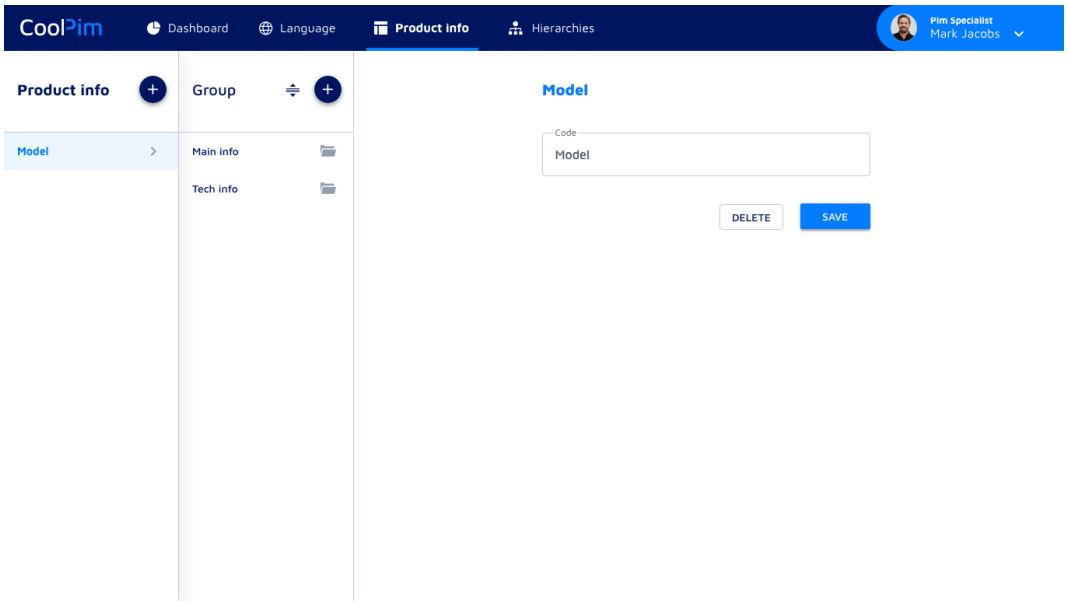


Figura 2.3 Sezione *Product Info*. Nella seconda colonna sono definiti i *Content Type Group* per quel prodotto specifico (in questo caso *Main info* e *Tech info*).

2.2.3 Le sezioni *Dashboard* e *Hierarchies*

Le rimanenti sezioni, *Dashboard* e *Hierarchies*, sono quelle in cui si è deciso di introdurre nuovi servizi e tipologie di dati che sfruttino l'*event sourcing pattern*.

Nella sezione *Hierarchies* sono visualizzate le liste di gerarchie già esistenti, ognuna delle quali può essere ulteriormente visualizzata nel dettaglio con informazioni aggiuntive per ogni nodo di cui sono composte. In questa sezione è possibile modificare gli attributi definiti nella struttura in fase di creazione del prodotto (i *Content Type*), propagando le modifiche ai nodi figli (o viceversa ereditando valori da

un nodo padre) ove indicato. È inoltre possibile (solo per alcune utenze) approvare e pubblicare i valori di tutti i *Content Type* appartenenti a uno stesso *Content Type Group*, come si vede in figura 2.4.

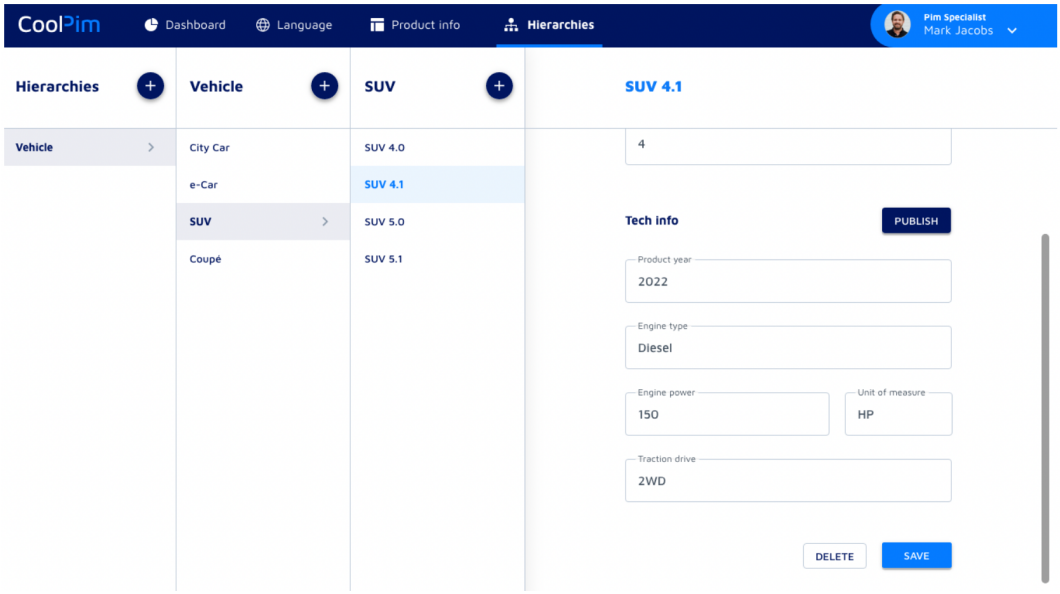


Figura 2.4 Sezione *Hierarchies*. Si veda in particolare la possibilità di modificare i campi e salvare le modifiche (tramite il bottone *Save*) e pubblicare un intero *Content Type Group* tramite il bottone *Publish*.

La sezione *Dashboard*, invece, visualizza alcune statistiche riguardante i prodotti, raggruppate secondo le seguenti metriche:

- Il numero di nodi totali
- Il numero di gerarchie, ossia di nodi radice
- Il numero di nodi foglia
- Il numero di nodi non completati, ossia che presentino almeno un *Content Type*, segnato come campo obbligatorio in fase di creazione del prodotto, privo di valore
- Il numero di nodi nuovi, ossia i cui *Content Type* non siano mai stati modificati
- Il numero di nodi bozza, ossia che abbiano almeno un loro *Content Type* modificato ma non abbiano ancora tutti i *Content Type Group* approvati

- Il numero di nodi pubblicati, ossia i cui *Content Type Group* siano stati tutti approvati almeno una volta

Il mio compito è stato quello di registrare gli eventi scatenati da un utente che modifichi i campi, approvi dei *Content Type Group*, modifichi la parentela tra nodi di una stessa gerarchia o elimini un nodo, per fornire una cronologia di modifiche consultabile sempre nella sezione *Hierarchies* e ricostruire, a partire dagli eventi registrati, un aggregato per popolare le statistiche mostrate nella sezione *Dashboard*, sfruttando i concetti dell'*event sourcing pattern* e le sue applicazioni con il CQRS *pattern*.

2.3 Tecnologie utilizzate

Il CoolPIM è stato sviluppato in React sfruttando la forte tipizzazione fornita da TypeScript e seguendo le linee guida e i componenti React forniti da Material UI.

Il DBMS utilizzato è MongoDB, che permette di modellare dei documenti che formano le collezioni seguendo uno schema definito tramite Mongoose, una libreria che permette la comunicazione tra MongoDB e l'ambiente di runtime TypeScript Node.js.

Il *query language* utilizzato è GraphQL, sfruttando Apollo Client, una *state management library* per TypeScript che permette di gestire i dati con GraphQL.

2.3.1 Esempio di gestione di una collezione di dati

Qualunque sia il tipo di dato che si vuole memorizzare nel database e col quale si vuole interagire, occorre definire tipi di dato e query, mettendo tutto in comunicazione sfruttando le tecnologie appena definite. Questa sequenza di operazioni è valida per i dati già presenti nell'applicazione ed è quella cui mi sono dovuto attenere definendo nuove collezioni di dati.

Queste operazioni sono:

- La definizione dello *schema* corrispondente ai documenti da inserire nella nuova collezione di MongoDB

- La definizione, nello *schema*, delle query di MongoDB che vengono chiamate a runtime sfruttando Apollo Client e la definizione delle stesse query GrahpQL
- La creazione del *model*, ossia della nuova collezione di MongoDB e delle query da esportare
- Parallelamente a queste operazioni, vengono definiti i tipi, quali quello corrispondente all'elenco di parametri da utilizzare nelle varie query e il valore ritornato dalle query (non necessariamente corrispondente al tipo del documento)

Capitolo 3

L'*event sourcing pattern*

L'*event sourcing pattern* fornisce un metodo meno classico per memorizzare i dati all'interno di un database: si basa sul concetto di eventi che vengono memorizzati uno alla volta in uno *store* di soli inserimenti.

L'essenza stessa dell'*event sourcing pattern*, lo rende perfetto per fornire un servizio di *auditing* e cronologia delle modifiche. Infatti, semplicemente interrogando il database che contiene gli eventi, ossia l'*event store*, è possibile recuperare ogni modifica che sia stata fatta dagli utenti (o almeno quelle che si è scelto di memorizzare) in ordine cronologico [4].

3.1 Che cos'è un evento

Prima di introdurre l'*event sourcing* e le sue applicazioni nel progetto CoolPIM, è necessario capire cosa sia un evento, ossia l'elemento principale di questo *pattern*.

Un evento rappresenta un fatto che si è verificato all'interno del dominio dell'applicazione. Gli eventi sono una fonte di verità e lo stato corrente è derivato da essi. Sono immutabili, e corrispondono alle attività svolte.

Un evento, in termini di *event sourcing*, generalmente contiene metadati unici come il *timestamp* dell'evento, l'identificatore univoco del soggetto, ecc. Il dato all'interno dell'evento viene usato nel *write model* per popolare lo stato attuale e prendere decisioni, così come per popolare i *read model*.

Ogni evento che si è verificato all'interno del dominio dell'applicazione è registrato nel database, in particolare negli *event store*, ossia la categoria di database incentrata in modo nativo sulla memorizzazione di eventi ed assimilato ad un registro di sole aggiunte.

Un *event store*, dunque, è un database diverso da quelli tradizionali: è pensato per memorizzare la storia dei cambiamenti, memorizzando gli eventi in ordine cronologico, aggiungendo nuovi eventi a quelli precedenti.

Gli eventi sono immutabili: non possono essere cambiati, ma i loro effetti possono essere alterati da eventi successivi. Ad esempio, consideriamo l'applicazione dell'*event sourcing pattern* a un eCommerce (esempio tipico in cui è preferita la rappresentazione ad eventi) e quindi un evento "FatturaEmessa" che sia stato aggiunto al log degli eventi. Supponiamo che l'indirizzo riportato nella fattura fosse sbagliato: un nuovo evento "FatturaAnnullata" verrà aggiunto e un successivo nuovo evento "FatturaEmessa", contenente l'indirizzo corretto, verrà aggiunto. Tutti e tre gli eventi sono memorizzati e continuano ad essere immutabili, ma il risultato finale è che un fattura è stata emessa con l'indirizzo corretto. Gli eventi sono immutabili, ma ciò non implica che il log degli eventi non possa cambiare [5].

3.2 La precedente versione del CoolPIM

La versione attuale del CoolPIM segue una precedente versione completamente *event sourced*: il database corrispondeva unicamente ad un *event store* in cui venivano registrati tutti gli eventi di interesse per poi costruire gli aggregati utilizzati per popolare il frontend dell'applicazione.

Tuttavia, essendo un'applicazione utilizzabile da più utenti contemporaneamente e, soprattutto, che deve poter essere utilizzata anche offline, sono sorti quasi immediatamente problemi di sincronizzazione sia tra le operazioni effettuate da diversi utenti che, localmente, effettuate da un singolo utente, alle cui modifiche non corrispondeva immediatamente un cambio dei valori. Viene infatti accumulato un ritardo tra le varie operazioni consecutive: l'aggiunta degli eventi all'*event store* da parte dell'applicazione, la pubblicazione degli eventi e i processi che li consumano. Inoltre, durante questo periodo, nuovi eventi che descrivono nuove e più recenti modifiche delle entità possono essere stati inviati all'*event store*.

Infine, un ulteriore complicazione che si ha nel creare un aggregato corrispondente al dato attuale, si ha nel momento in cui occorre aggiornare un'entità su cui è stato annullato un cambiamento: poiché gli eventi sono per definizione immutabili, non devono mai essere aggiornati, e occorre quindi in questo caso aggiungere all'*event store* un evento che compensi quello da annullare, andando a impattare sulle dimensioni dell'*event store* e sulle prestazioni nel calcolo dell'aggregato [4].

Questi problemi sono intrinsechi alla natura dell'*event sourcing pattern* stesso e la soluzione adottata dall'azienda è stata dunque quella di distribuire una nuova versione basata unicamente su un database non relazionale, ossia noSQL, seguendo un tradizionale approccio CRUD (*Create*, *Read*, *Update* e *Delete*).

3.2.1 L'approccio CRUD

Nel caso tradizionale dell'approccio CRUD, il tipico modo di processare i dati prevede di leggere i dati, effettuare alcune modifiche e aggiornare lo stato corrente del dato con nuovi valori.

Questo approccio ha alcune limitazioni:

- I sistemi CRUD effettuano operazioni di aggiornamento direttamente sul *data store*, il che può ridurre le performance e la reattività, limitando la scalabilità a causa del sovraccarico di elaborazione richiesta
- In un ambiente collaborativo con più utenti contemporaneamente attivi, conflitti di aggiornamento dei dati sono più probabili poiché le operazioni di aggiornamento avvengono su un singolo oggetto dei dati
- A meno che non sia presente un addizionale meccanismo di *auditing* che registri nel dettaglio ogni singola operazione, non viene tenuta traccia della cronologia delle modifiche [4]

Per questi motivi è stato deciso di reintrodurre l'*event sourcing pattern* ma solo per alcune tipologie di dato e per fornire un servizio di *auditing*, adottando dunque una soluzione ibrida, in parte basata sul tradizionale CRUD e in parte *event sourced*.

3.2.2 I vantaggi dell'*event sourcing pattern*

Gli eventi registrati nell'*event store* vengono quindi utilizzati esclusivamente per fornire servizi in lettura quali *auditing* e creazione degli aggregati (ripetendo e consumando gli eventi legati a una certa entità) affinché vengano calcolate le statistiche da mostrare nella sezione *Dashboard*.

Dunque, è stato necessario innanzitutto creare un nuovo database a eventi, metterlo in comunicazione con l'applicazione e, soprattutto, considerare eventuali problemi di prestazione, in quanto, la memorizzazione di un evento corrispondente a ogni modifica effettuata da (potenzialmente) più utenti contemporaneamente, si traduce in una collezione di dati notevolmente ampia da interrogare ogniqualvolta l'utente ne faccia richiesta.

3.3 L'*event sourcing pattern* applicato all'*auditing*

L'idea principale è quella di realizzare una collezione di eventi che idealmente contenga le informazioni essenziali, come mostrato in figura 3.1.

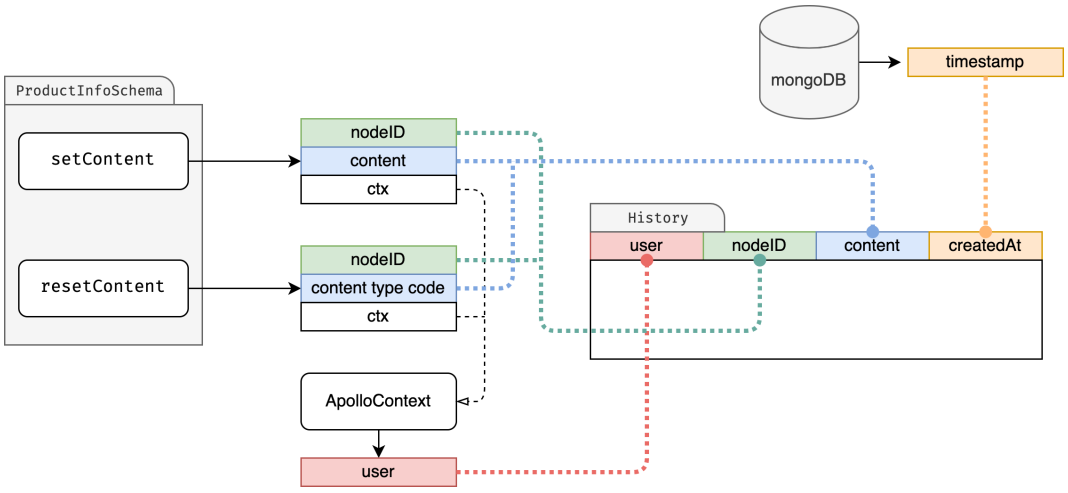


Figura 3.1 Schema concettuale per la realizzazione di una nuova collezione *History*. Viene messo in risalto il modo in cui recuperare le informazioni essenziali, ossia l'utente, l'id del nodo di cui si vuole registrare un evento, il contenuto che è stato modificato e la data in cui è avvenuto l'evento.

La cattura di questi eventi deve necessariamente avvenire a seguito di una corrispondente azione da parte di un utente. Per questo motivo, l'idea è stata quel-

la di sfruttare dei metodi già presenti, ossia `setContent` e `resetContent`, che aggiornano i dati con la nuova informazione inserita dall'utente.

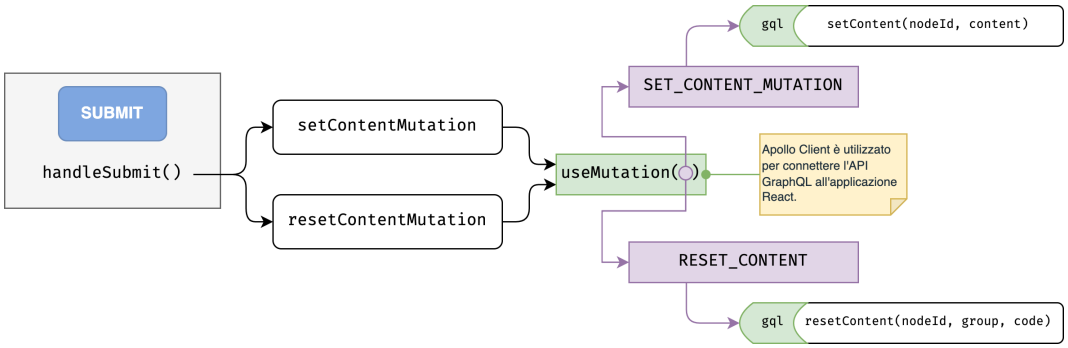


Figura 3.2 Schema concettuale che identifica il flusso seguito dall'applicazione al click da parte di un utente sul bottone *Submit* dopo aver effettuato le modifiche. L'idea è quella di chiamare dei nuovi metodi che registrino gli eventi all'interno dei metodi `setContent` e `resetContent` che, a loro volta, effettuano le query GraphQL che aggiornano il dato vero e proprio (definite *mutation*).

3.3.1 La collezione *History*

Prima di definire come memorizzare gli eventi e come recuperarli quando necessario, occorre decidere quali informazioni registrare e in seguito a quali azioni.

Si è dunque deciso di registrare gli eventi legati alle principali operazioni attuabili dalla sezione *Hierarchies*, ossia:

- Modificare i *Content Type* di un nodo
- Approvare un *Content Type Group* di un nodo
- Modificare le parentele tra nodi di una stessa gerarchia
- Creare un nuovo nodo o gerarchia
- Eliminare un nodo (ed eventuali suoi figli in modo ricorsivo)

Il progetto CoolPIM utilizza MongoDB, tramite il modellatore di oggetti Mongoose, che permette di realizzare soluzioni basate su schemi per modellare i dati [6]. Ho realizzato la collezione di dati corrispondenti agli eventi, chiamata *History*, seguendo il modello di altre collezioni già esistenti nel progetto, sfruttando la creazione di *Model* e *Schema* fornita da Mongoose.

3.3.2 Definizione dei tipi

Nel file TypeScript `history.ts` sono stati specificati tutti i tipi utili per definire l'entità principale `history`, che deve contenere tutte le informazioni utili di un evento, e i metodi di comunicazione (in scrittura e in lettura) con il database.

In particolare sono stati definiti il tipo `HistoryS`, `HistoryQueryResult` e `History`.

Il tipo `HistoryS` include tutte le informazioni utili da registrare e fornire poi all'utente sotto forma di cronologia di modifiche.

```
type HistoryS = {
    _id: HistoryId;
    productInfo: ProductInfoId | null;
    node: NodeId | null;
    group: GroupCode | null;
    code: ContentTypeCode | null;
    content: DistributiveOmit<ContentS, 'group' | 'code'> | null;
    userName: string | null;
    createdAt: Date;
    status: NodeHistoryStatus | null;
    parent: NodeId | null;
    oldParent: NodeId | null;
    ancestors: NodeId[] | null;
};
```

Figura 3.3 Il tipo `HistoryS`. Molti parametri possono avere un valore nullo poiché la loro presenza nell'entità evento dipende dal tipo di evento cui corrisponde.

In particolare comprende informazioni generiche dell'evento, come si vede in figura 3.3:

- `_id`: il GUID (un identificatore unico globale, automaticamente generato da MongoDB) della *entry* corrispondente all'evento nella collezione
- `productInfo`: il GUID del prodotto cui fa riferimento l'evento
- `node`: il GUID del nodo cui fa riferimento l'evento
- `group`: il codice dell'eventuale *Content Type Group* (definito in fase di creazione del prodotto) in cui è avvenuta una modifica o un'approvazione

- `code`: il codice dell'eventuale *Content Type* che è stato modificato
- `content`: l'eventuale contenuto che è stato modificato, includendo sia il tipo di contenuto che il suo nuovo valore
- `userName`: il nome dell'utente che ha scatenato l'evento
- `createdAt`: la data in cui l'evento è stato registrato

Oltre a informazioni riguardanti azioni specifiche eseguite dall'utente, in particolare:

- `status`: un valore che distingue l'evento come evento in cui è stato approvato un *Content Type Group* (e quindi tutti i relativi *Content Type* definiti al suo interno)
- `parent`: in caso di creazione o cambio di gerarchia di un nodo, il GUID corrispondente al nodo padre (eventualmente quello nuovo)
- `oldParent`: in caso di cambio di gerarchia di un nodo, il GUID corrispondente al precedente nodo padre
- `ancestors`: la lista di tutti i GUID degli antenati del nodo corrente

```
type HistoryQueryResult = {
    data: (HistoryS | null)[];
    totalCount: number;
};
```

Figura 3.4 Il tipo `HistoryQueryResult`. `data` ha valore nullo nel caso in cui la query eseguita non abbia alcuna corrispondenza e in tal caso `totalCount` varrebbe `0`.

Il tipo `HistoryQueryResult`, corrisponde al tipo ritornato dalle query in lettura sul database, e comprende i dati (del tipo `HistoryS` appena descritto) e un valore `totalCount`, corrispondente al numero totale di documenti ritornati dalla query, informazione necessaria per l'implementazione, avvenuta in un secondo momento, della paginazione.

Il tipo `History`, infine, identifica il tipo di oggetto da utilizzare per interrogare il database, a seconda del raggio di ricerca. I parametri possibili sono quelli mostrati in figura 3.5:

- `node`: il GUID del nodo che si vuole ricercare
- `group`: il codice del *Content Type Group* da ricercare
- `code`: il codice del *Content Type* da ricercare
- Tre valori numerici (`offset`, `limit` e `direction`) anche questi introdotti successivamente per implementare la paginazione

```
type History = {
    node: NodeId;
    group?: GroupCode;
    code?: ContentTypeCode;
    offset: number;
    limit: number;
    direction: number;
};
```

Figura 3.5 Il tipo `History`. I parametri `group` e `code` sono opzionali in quanto la query può essere eseguita a livello di nodo o, in modo più stretto, a livello di gruppo o contenuto, specificando i rispettivi codici.

3.3.3 Definizione del *Model* e dello *Schema*

Il passo successivo è stato quello di definire uno *Schema*, una struttura di Mongoose che permette di mappare una collezione di MongoDB, definendo la forma dei documenti di quella collezione, sfruttando i tipi appena descritti [6]. In questo modo è possibile indicare i campi obbligatori di un documento e le query da esportare al frontend dell'applicazione, attraverso GraphQL. In questo caso particolare, l'unica query da esporre è quella che permette di leggere un evento, definendo quindi gli argomenti da utilizzare per effettuare la query, ossia quelli descritti dal tipo `History`.

Parallelamente allo *Schema*, è stato definito anche il *Model*, ossia un costruttore compilato a partire dalla definizione dello *Schema*, responsabile della scrittura e lettura dei documenti dal sottostante database di MongoDB [6].

Nel *Model* è stato definito non solo il metodo per recuperare una lista di determinati eventi, ma soprattutto i metodi per aggiungere degli eventi alla collezione, in particolare:

- Il metodo `addHistory`, per aggiungere un evento di modifica di un contenuto
- Il metodo `addApproveHistory`, per aggiungere un evento di approvazione di un *Content Type Group*
- Il metodo `addCreateNodeHistory`, per aggiungere un evento di creazione di un nodo o di una nuova gerarchia
- Il metodo `addChangeParentHistory`, per aggiungere un evento di cambio di parentela tra nodi di una stessa gerarchia
- Il metodo `addRemoveNodeHistory`, per aggiungere un evento di rimozione di un nodo e, ricorsivamente, anche per tutti i suoi eventuali figli

Questi metodi non sono stati esposti al frontend, in quanto si è scelto di utilizzarli direttamente all'interno di altri *Model*, in particolare quello che gestisce le scritture delle stesse operazioni di cui si è scelto di registrare gli eventi (modifica, approvazione, cambio di parentela, creazione e rimozione).

Capitolo 4

Studio delle prestazioni di lettura degli eventi dalla collezione *History*

L'*event sourcing pattern* permette di avere ottime prestazioni in fase di scrittura, in quanto si tratta di semplici aggiunte ad una collezione. Viceversa, l'operazione di lettura di molteplici documenti può risultare lenta e, nel caso specifico della sua applicazione al CoolPIM, gli eventi da registrare (utilizzati successivamente per popolare la cronologia delle modifiche) sono numerosi ed è quindi necessario che il tempo di realizzazione di una lettura di questi dati non risulti eccessivo.

Durante lo sviluppo di questa funzionalità di *auditing*, sono state provate diverse soluzioni, in alcuni casi anche affiancate da misure e test specifici, per garantire un servizio veloce e non bloccante.

4.1 Differenza tra `useQuery` e `useLazyQuery`

Il client Apollo, in combinazione con il *query language* GraphQL, entrambi utilizzati nel progetto per mettere in comunicazione frontend e database, permette di eseguire una specifica query tramite l'hook `useQuery`. In questo modo, non appena un componente React viene renderizzato, se è stata definita una query tramite questo hook, questa viene eseguita e il suo valore memorizzato per essere utilizzato, ad esempio, dal componente in questione [7].

Una prima soluzione ha previsto l'utilizzo di `useQuery` per recuperare le informa-

zioni riguardanti il nodo, il gruppo o il contenuto specifico della pagina visualizzata, al caricamento della pagina stessa. Per quanto detto finora si tratta di una soluzione poco ottimale poiché, nel caso di una collezione altamente popolata, le query avrebbero impiegato più tempo a risolversi e di conseguenza la pagina avrebbe richiesto molto più tempo per renderizzare tutti i suoi componenti.

A fronte di questo problema, sono state realizzate in successione due diverse soluzioni: in un primo momento si è deciso di utilizzare un altro hook fornito da Apollo, chiamato `useLazyQuery`; successivamente si è deciso invece di spostare l'utilizzo dell'hook `useQuery` all'interno del componente che avrebbe poi dovuto renderizzare la cronologia delle modifiche.

Vediamo nel dettaglio le due soluzioni e il perché si sia optato per la seconda, benché simili.

4.1.1 La soluzione con `useLazyQuery`

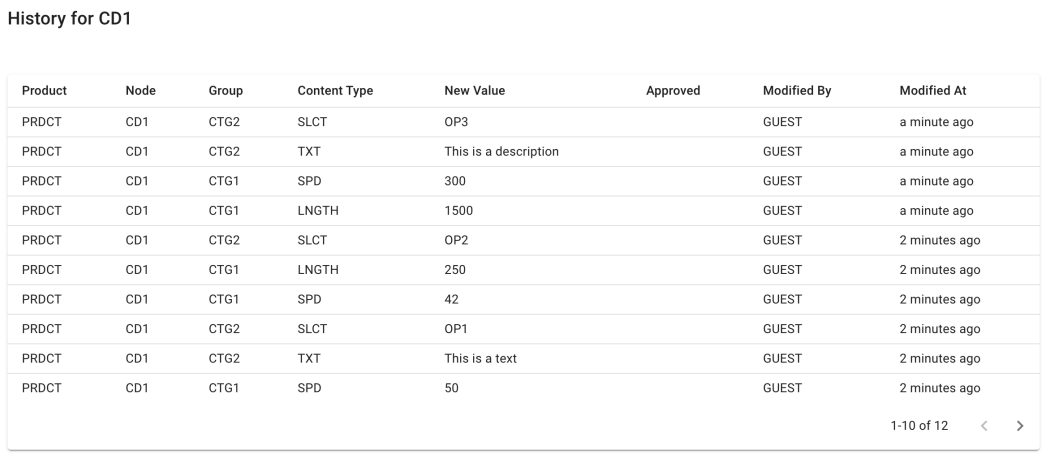
Ciò che contraddistingue l'hook `useLazyQuery`, è la possibilità di eseguire query in risposta a determinati eventi. A differenza di `useQuery`, dunque, `useLazyQuery` non esegue immediatamente la query associata, ma ritorna una *query function* pronta per essere chiamata quando necessario. In questo caso specifico, per risolvere i problemi di caricamento di tutta la pagina, sarebbe stato l'evento di click sul bottone che mostra la cronologia ad eseguire le query interessate.

Questa prima soluzione si è dimostrata efficace, ma rimane irrisolto il problema di lettura degli eventi interessati nel caso in cui la collezione avesse raggiunto dimensioni ragguardevoli.

4.1.2 La soluzione con `useQuery` e la paginazione

Questo problema trova soluzione nell'utilizzo della paginazione, che in questo caso risulta perfettamente calzante, poiché la cronologia è visualizzata utilizzando una tabella (come quella mostrata in figura 4.1), organizzata per l'appunto in pagine. Tuttavia, per visualizzare correttamente la tabella e far sì che tutte le sue funzioni siano corrette, è necessario avere già i dati restituiti dalla query, prima che la tabella venga renderizzata per la prima volta. Per questo motivo, la soluzione più efficace si è rivelata essere quella di spostare l'utilizzo dell'hook `useQuery` all'interno del modale

in cui viene renderizzata la tabella: in questo modo, la query sarebbe stata eseguita in automatico all'apertura del modale, senza impattare sui tempi di caricamento della pagina, ottenendo lo stesso risultato garantito dall'uso dell'hook `useLazyQuery`.


History for CD1

Product	Node	Group	Content Type	New Value	Approved	Modified By	Modified At
PRDCT	CD1	CTG2	SLCT	OP3		GUEST	a minute ago
PRDCT	CD1	CTG2	TXT	This is a description		GUEST	a minute ago
PRDCT	CD1	CTG1	SPD	300		GUEST	a minute ago
PRDCT	CD1	CTG1	LNGTH	1500		GUEST	a minute ago
PRDCT	CD1	CTG2	SLCT	OP2		GUEST	2 minutes ago
PRDCT	CD1	CTG1	LNGTH	250		GUEST	2 minutes ago
PRDCT	CD1	CTG1	SPD	42		GUEST	2 minutes ago
PRDCT	CD1	CTG2	SLCT	OP1		GUEST	2 minutes ago
PRDCT	CD1	CTG2	TXT	This is a text		GUEST	2 minutes ago
PRDCT	CD1	CTG1	SPD	50		GUEST	2 minutes ago

1-10 of 12

Figura 4.1 Esempio di tabella che mostra la cronologia di modifiche di un nodo.

4.2 La paginazione

Non è raro che in alcune applicazioni vi sia la necessità di mostrare liste di dati fin troppo numerosi per essere elaborati o visualizzati immediatamente. Come già detto, questo problema è riscontrato nel CoolPIM per la visualizzazione della cronologia delle modifiche e la paginazione è la soluzione più comune a questo tipo di problema, permettendo di caricare piccole quantità di dati poco alla volta, solo quando necessario ed esplicitamente richiesto da un'azione dell'utente [8].

4.2.1 Paginazione a livello di query

MongoDB mette a disposizione alcuni metodi molto semplici che permettono di implementare la paginazione, concatenandoli ad altri metodi di risoluzione di una query. Si tratta dei metodi `skip` e `limit` [9]. Questi due metodi accettano come parametro un numero intero che indica, per il primo, il numero di documenti da saltare nell'insieme dei risultati, mentre, per il secondo, il numero massimo di documenti che la query deve ritornare.

Applicando questi metodi per implementare la paginazione, avremo che il numero della pagina selezionata moltiplicato per il valore di `limit` indicherà il numero di documenti da saltare quando verrà eseguita la query, mentre il valore di `limit` indicherà il numero di documenti contenuto in ogni pagina [10].

Ad esempio, per un valore di `limit` pari a `10`, trovandoci alla pagina `0` (gli indici di pagina partono dal numero `0`) avremo che il numero di documenti da saltare sarebbe `0 * limit` e quindi `0`. Trovandoci alla pagina `1`, invece, il numero di documenti da saltare sarebbero `1 * limit` e quindi `10`, e così via, mostrando sempre un numero di eventi pari a `limit`.

Questa soluzione, inoltre, sfrutta la cache fornita da Apollo Client. Questo infatti memorizza i risultati delle query GraphQL in una cache locale, normalizzata e *in-memory*, garantendo delle risposte pressoché immediate a delle query corrispondenti a dati già memorizzati nella cache, senza bisogno di inviare una nuova richiesta di rete. Questo si traduce in un notevole vantaggio, poiché le query sono eseguite ogniqualvolta si scorra in avanti o indietro l'elenco di pagine degli eventi: in questo modo, tornando a una pagina per cui è già stata effettuata precedentemente una query, il risultato di questa è già disponibile nella cache, popolando immediatamente la pagina senza ulteriori richieste di rete. Un esempio di questo funzionamento è illustrato in figura 4.2.

Ricorrere a questa soluzione fornisce dunque prestazioni decisamente migliori rispetto ad ottenere tutti i documenti e doverli poi filtrare in fase di esecuzione per visualizzarli correttamente.

4.2.2 Paginazione implementata con l'utilizzo degli indici

Per avere tempi di risposta ancora migliori, è possibile utilizzare un'ulteriore struttura dati fornita da MongoDB, ossia gli indici, che permettono di effettuare le cosiddette *range queries* [9].

Gli indici di MongoDB

Gli indici sono strutture dati particolari che contengono una piccola parte dell'insieme di dati della collezione in una forma facile da attraversare. Un indice memorizza il valore di un campo specifico (o di un insieme di campi), ordinati per il valore

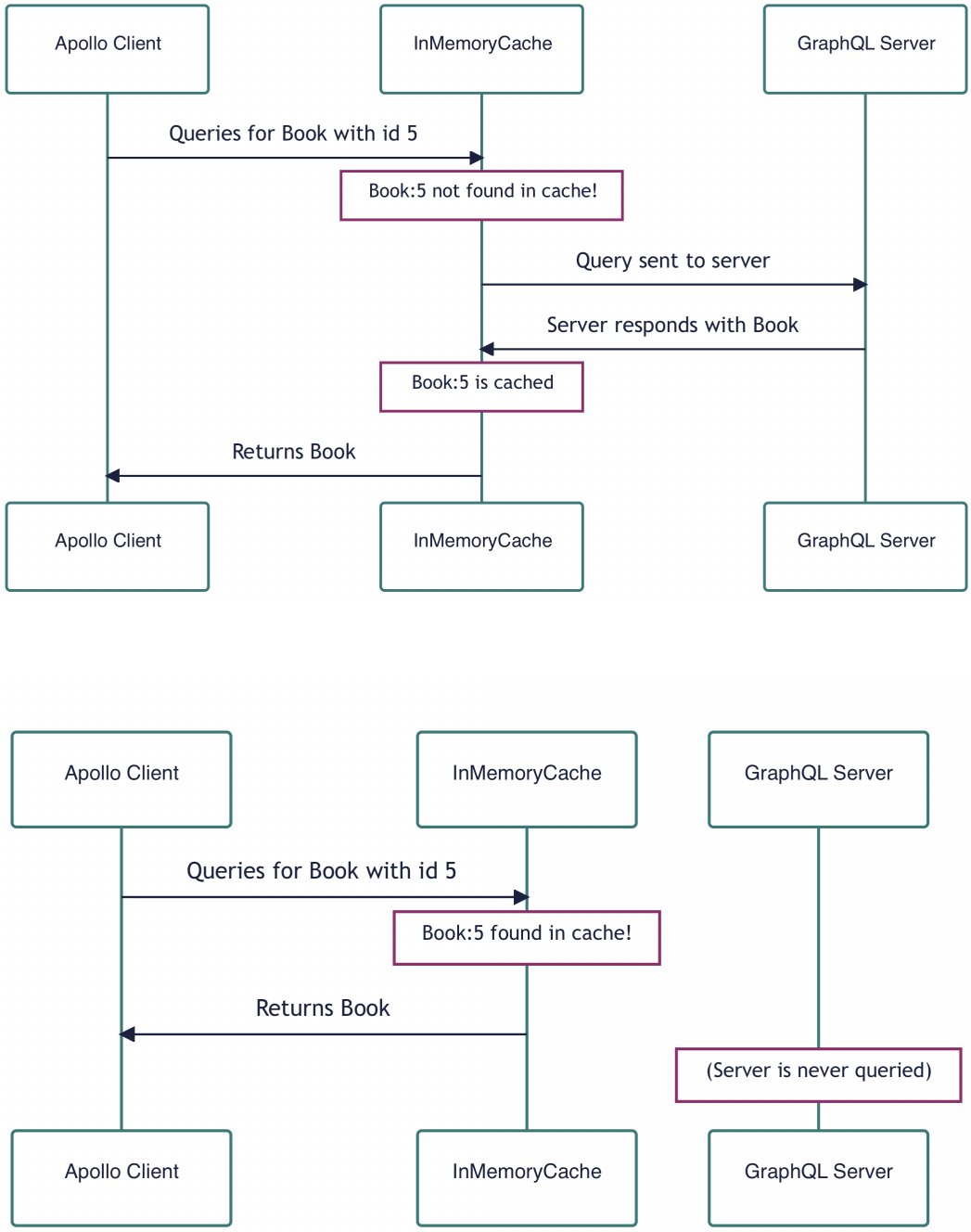


Figura 4.2 Esempio di funzionamento della cache fornita da Apollo Client. Se un risultato non viene trovato nella cache, si esegue la query e si memorizza il risultato nella cache. In questo modo, successive query che richiedono lo stesso risultato saranno risolte tramite l'utilizzo della cache, senza chiamare in causa il server.

di questo campo. Questo ordinamento permette di avere corrispondenze di uguaglianza efficienti e operazioni di query basate sull'intervallo (*offset*), sostituendo di fatto l'operazione svolta dal metodo `skip` e con performance migliori al crescere dell'*offset*.

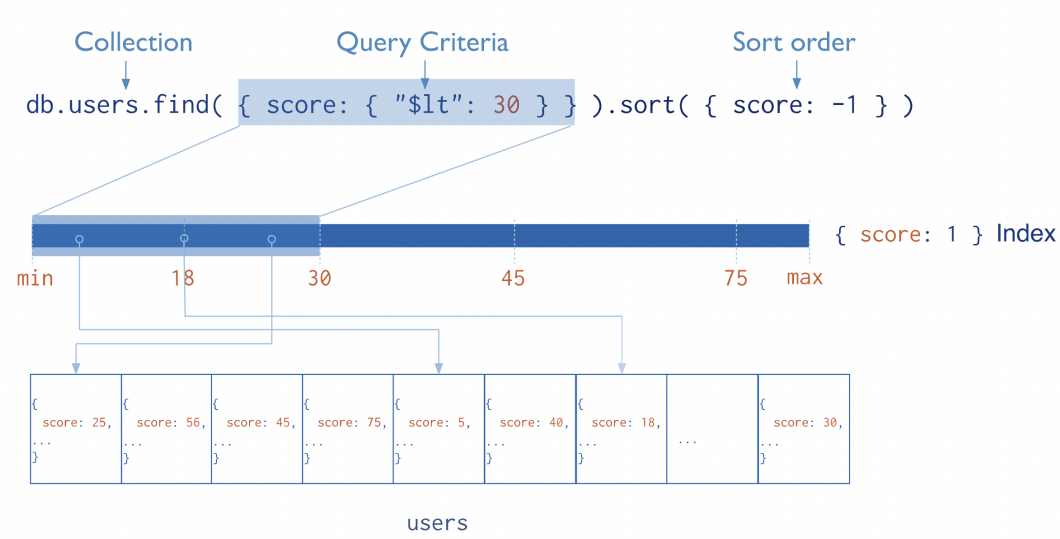


Figura 4.3 Diagramma che illustra il funzionamento di una query che seleziona ed ordina i documenti utilizzando un indice.

Senza l'utilizzo degli indici, MongoDB dovrebbe effettuare una scansione di tutta la collezione, ossia per ogni documento presente in essa, per poi selezionare solo i documenti che combaciano con la query definita. Definendo invece un opportuno indice per quella specifica query, MongoDB lo può utilizzare per limitare il numero di documenti da ispezionare.

L'utilizzo degli indici, dunque, fa si che vengano scansionati unicamente i documenti interessati [11].

Le *range queries*

Le *range queries* possono usare gli indici per evitare di scansionare documenti non necessari.

Per implementare la paginazione con le *range queries* occorre:

- Scegliere un campo che cambi in modo consistente col tempo e che sia unico (per evitare duplicati), corrispondente all'*offset*

- Interrogare i documenti il cui campo sia minore o maggiore (a seconda dell'ordinamento) del valore da confrontare.

L'ultimo valore va quindi memorizzato per poter essere poi utilizzato come valore di confronto per le successive query.

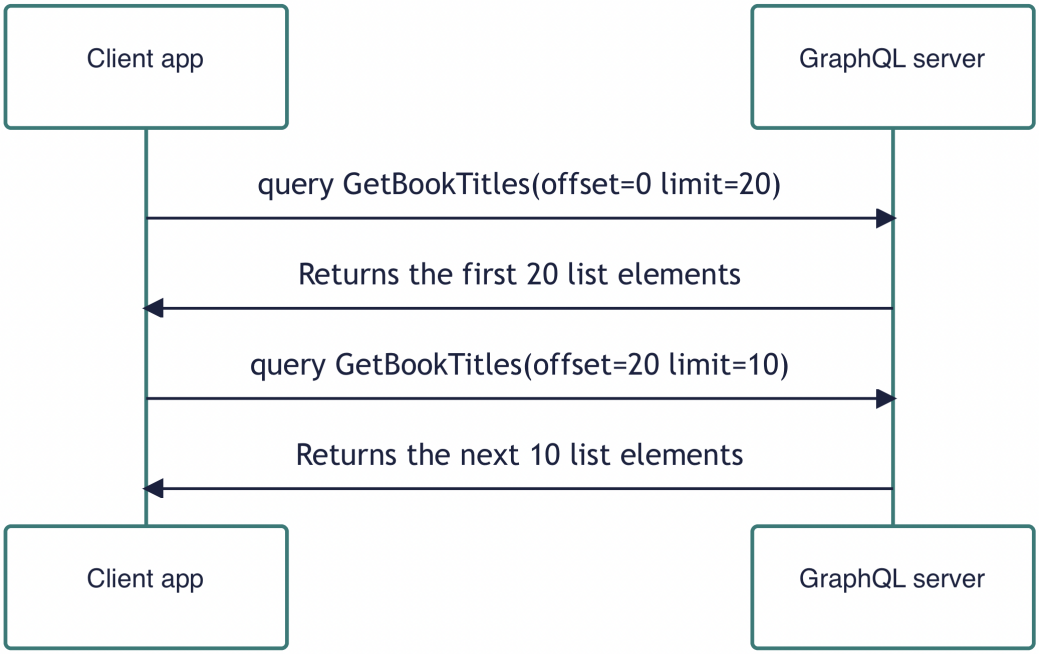


Figura 4.4 Esempio di funzionamento di query GraphQL basate sulla paginazione, tramite la definizione di *offset* e *limit*.

4.2.3 Paginazione tramite *range queries*

Nella versione ultimata del metodo `getHistory`, ossia quello utilizzato per ottenere i documenti degli eventi relativi a uno specifico nodo, gruppo o contenuto, il campo utilizzato come `offset` (ovvero come valore precedentemente usato col metodo `skip`) è quello della data in cui è stato registrato l'evento. A seconda che si stia scorrendo la tabella in avanti o indietro (indicato nel codice col parametro `direction`) verrà utilizzata la data dell'evento in testa (se si sta andando indietro) o in coda (se si sta andando in avanti), e quindi filtrando i documenti opportunamente per date maggiori o minori di quella corrente.

Come `limit` viene invece semplicemente utilizzato il numero di eventi che si vuole mostrare in una pagina della tabella (per impostazione predefinita pari a

10). Il risultato finale è la query mostrata in figura 4.5. Questa viene utilizzata in combinazione con gli altri metodi forniti da MongoDB per ottenere la lista di documenti desiderata.

Poichè i criteri di ricerca dipendono dalla data e da una combinazione di codice del nodo, codice del gruppo e codice del contenuto, i campi utilizzati dalla query sono: `createdAt`, `node`, `group`, `code` e il campo `_id` (ossia il GUID corrispondente al singolo evento) che per definizione di creazione di un indice, deve essere presente. Per questo motivo, sono stati definiti tre diversi indici nel modo seguente:

```
schema.index({ node: 1, createdAt: -1, _id: -1 });

schema.index({ node: 1, group: 1, createdAt: -1, _id: -1 });

schema.index({ node: 1, group: 1, code: 1, createdAt: -1, _id: -1 });
```

I valori numerici (`1` e `-1`) indicano l'ordinamento da utilizzare per quel campo specifico (rispettivamente crescente e decrescente), ma MongoDB è in grado comunque di utilizzare lo stesso indice precedentemente definito anche per combinazioni di ordinamenti differenti.

La differenza nella ricerca dei documenti con o senza l'utilizzo degli indici è evidente nell'esempio seguente, nel numero di documenti scansionati per la ricerca a fronte dell'esecuzione della stessa query per ricercare gli eventi di un nodo specifico.

In questo esempio sono state riportate le `executionStats` dell'*explain plan* riguardante la query eseguita con MongoDB.

L'unico inconveniente che può derivare dall'uso degli indici è che è richiesto dello spazio per memorizzarli nel database. Tuttavia, in questo caso specifico, per il numero di eventi che si è previsto registrare al massimo (ossia intorno ai 100.000 eventi), gli indici occuperebbero in totale qualche centinaio di chilobyte, una dimensione più che accettabile in questo utilizzo.

Oltretutto, questa soluzione non può sfruttare la cache fornita da Apollo Client, in quanto il suo funzionamento è legato alla riconoscibilità della query e non del suo risultato. Ciò significa che se due query definite in modo diverso, forniscono però lo stesso risultato, questo verrà memorizzato due volte nella cache anziché una sola per essere poi fornito come risultato all'altra query. Questa situazione

```
const query = {
    node: node,
    ...(group ? { group: group } : {}),
    ...(code ? { code: code } : {}),
    $or: [
        { createdAt: direction ?
            { $lt: new Date(offset) } :
            { $gt: new Date(offset) }
        },
        {
            ...(id
                ? { _id: direction ?
                    { $lt: id } :
                    { $gt: id },
                createdAt: { $eq: new Date(offset) } }
                : {})
        }
    ]
};

historyModel
    .find(query)
    .sort(
        { createdAt: direction ? -1 : 1,
        _id: direction ? -1 : 1
    })
    .limit(limit)
    .lean()
    .exec();
```

Figura 4.5 Definizione di *range query*. Tramite l'operatore `$or` si ricercano i documenti precedenti o successivi all'`offset` (a seconda della direzione in cui si stanno sfogliando le pagine) o i documenti il cui valore `createdAt` combaci con quello dell'`offset` (ciò è possibile per modifiche multiple che vengono salvate contemporaneamente) e abbia un `_id` minore o maggiore di quello dell'evento utilizzato come *offset* (poiché gli eventi sono ordinati secondo la data di creazione e secondo il loro `_id`, in questo modo si è certi di avere come risultato sempre gli stessi documenti e sempre nello stesso ordine).

```
"executionStats": {
    "executionSuccess": true,
    "nReturned": 8,
    "executionTimeMillis": 0,
    "totalKeysExamined": 0,
    "totalDocsExamined": 51,
    "executionStages": {...},
    "allPlansExecution": []
}
```

```
"executionStats": {
    "executionSuccess": true,
    "nReturned": 8,
    "executionTimeMillis": 0,
    "totalKeysExamined": 8,
    "totalDocsExamined": 8,
    "executionStages": {...},
    "allPlansExecution": [...]
},
```

Figura 4.6 Statistiche di esecuzione della query senza e con gli indici. Si noti la differenza nel numero di documenti totali esaminati.

si verifica in questo caso in quanto, a differenza della precedente soluzione in cui nella query venivano fornite la pagina e il numero di elementi da restituire (dati che non cambiavano sia che si stesse sfogliando le pagine in avanti o indietro), le query differiscono a seconda che si stiano sfogliando le pagine degli eventi in avanti o indietro. Differiscono in quanto la paginazione è basata su un intervallo di documenti il cui valore di `createdAt` deve essere maggiore o minore dell'`offset`, a seconda della direzione in cui si stanno sfogliando le pagine specificata dal valore di `direction`. Così facendo, i dati contenuti in una pagina di eventi corrispondono al risultato di due diverse query.

Ad esempio se ci trovassimo a pagina 2 dell'elenco di eventi e volessimo caricare pagina 3, verrebbe eseguita una query che ricerchi gli eventi precedenti all'ultimo evento in elenco. Se invece fossimo a pagina 4 e volessimo caricare pagina 3, verrebbe eseguita una query che ricerchi gli eventi successivi al primo evento in elenco. In entrambi i casi il risultato è il medesimo, ma le query differiscono per il valore dell'`offset` (ossia la data di riferimento con cui filtrare i documenti) e per il valore di `direction`, che specifica il tipo di confronto da effettuare (specificato dagli operatori `$lt` e `$gt`). Ovviamente, dovessero essere eseguite nuovamente queste stesse query, la cache verrebbe comunque sfruttata.

Questa soluzione dunque non sfrutta a pieno le potenzialità della cache fornita da Apollo Client, dovendo in sostanza risolvere una query in più del dovuto affinché non vengano eseguite ulteriori richieste al server. Tuttavia, come dimostrato nel successivo capitolo 5, questa soluzione offre tempi di risposta medi decisamente migliori, non solo rispetto a una soluzione che non adotti la paginazione, ma anche

nei confronti della soluzione precedente a questa, con paginazione implementata tramite MongoDB sfruttando i metodi `skip` e `limit`.

Capitolo 5

Misurazioni e *stress test* delle prestazioni di lettura degli eventi

Le soluzioni presentate nel capitolo precedente sono state sottoposte a misurazioni e test che arrivassero a coprire anche i casi più estremi, per studiarne il comportamento al crescere della dimensione dell'*event store*, mettendo poi a confronto queste prestazioni tra loro e con il caso più semplice, che non sfrutta la paginazione.

Sono stati presi in considerazione per i test non solo la dimensione della collezione di dati, che è stata fatta variare da centinaia di documenti a centinaia di migliaia (considerato dall'azienda il numero massimo di documenti per questo tipo di applicazione e il suo utilizzo), ma anche il numero di utenti contemporaneamente attivi che operassero sui dati, considerando una decina di utenti come dimensione realistica e un centinaio di utenti come caso estremo.

5.1 Il framework Locust

Questi test sono stati effettuati per mezzo di un framework apposito, chiamato Locust, che permette di simulare letture (e scritture) su un database, specificando il numero di utenti attivi e la frequenza delle chiamate, come mostrato in figura 5.1.

Queste simulazioni vengono gestite tramite script dedicati scritti in Python: in un apposito file, chiamato `locustfile.py` viene definita una classe `WebsiteUser` (a partire da una classe `HttpUser` importata da Locust) in cui si definiscono le task,

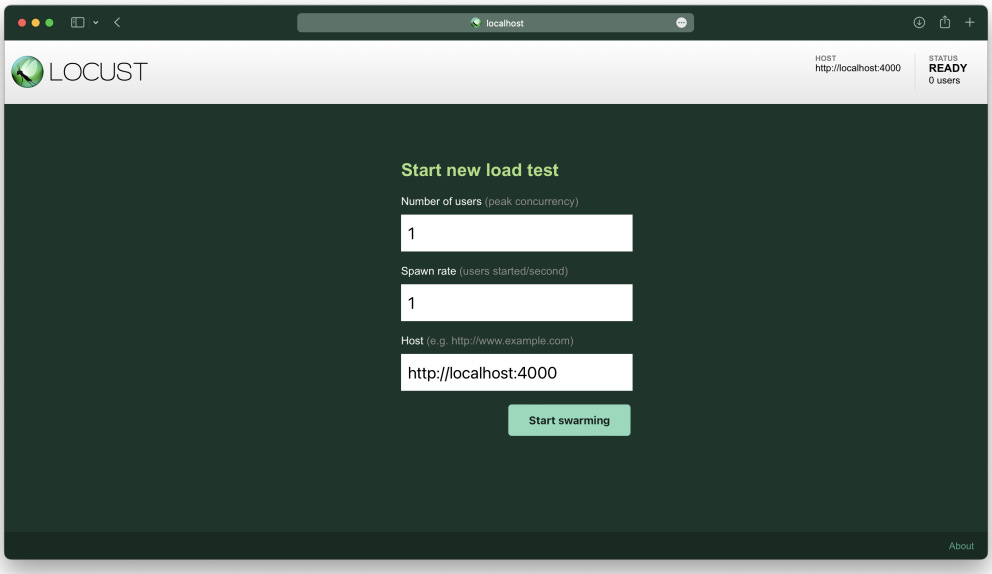


Figura 5.1 Landing page raggiungibile alla porta 8089 dell'indirizzo locale dopo aver eseguito il comando `locust` nel direttorio in cui è presente lo script `locustfile.py` che definisce la connessione e i task da eseguire. Da qui è possibile specificare il numero di utenti massimi attivi, e il numero di utenti che vengono creati al secondo (*spawn rate*). I test sono stati effettuati con 10 e 100 utenti e rispettivamente 10 e 100 utenti creati al secondo (in questo modo al primo secondo viene immediatamente raggiungo il picco massimo e si può vedere l'andamento nei secondi successivo a carico massimo).

ossia le operazioni che verranno svolte da un utente fittizio, e l'host corrispondente al web server (nel caso di Apollo Client, in ascolto sulla porta 4000 dell'indirizzo locale), come si vede in questo esempio:

```
class WebsiteUser(HttpUser):
    tasks = [UserBehavior]
    host = "http://localhost:4000"
```

5.1.1 L'attributo `wait_time`

È anche possibile definire l'attributo `wait_time` che permette di introdurre dei ritardi dopo l'esecuzione di ogni task. Se nessun `wait_time` viene specificato, il task successivo viene eseguito non appena un altro termina.

Il `wait_time` può essere definito tramite opportuni metodi esportati dalla libreria `locust`, quali:

- `constant`, permette di definire una quantità di tempo fissa
- `between`, definisce un range di valori entro cui viene scelto, ogni volta in modo casuale, il tempo di attesa tra un task e l'altro
- `constant_throughput`, definisce un tempo adattivo che assicuri che vengano eseguiti al massimo `X` task al secondo
- `constant_pacing`, definisce un tempo adattivo che assicuri che venga eseguito un task al massimo una volta ogni `X` secondi (l'operazione inversa di `constant_throughput`)

L'utilizzo di `constant_throughput` permette, specificando il numero di utenti operativi, di definire il numero di task da eseguire al secondo al picco massimo. Ad esempio, se si volessero eseguire 500 task al secondo, basterebbe definire un `constant_throughput` di 0.1 secondi con un numero di utenti pari a 5000 [12].

Gli script sono stati scritti utilizzando diversi tipi di `wait_time`, in particolare definendolo tramite il metodo `between`, permettendo di effettuare simulazioni con un numero di letture al secondo concorrenti realistico, anche con centinaia di utenti. Tuttavia, si è preferito stressare al massimo l'applicazione simulando un elevato numero di operazioni al secondo, senza definire di conseguenza alcun wait_time.

5.2 Test in scrittura

Una prima parte di test riguarda la popolazione della collezione di dati con un numero crescente di documenti. È stato deciso di definire tre tipi di prodotti (denominati in modo fittizio e riconoscibile come *car*, *truck* e *boat*) per ognuno dei quali sono stati definiti dei *Content Type Group* e, di conseguenza, dei *Content Type*, ed è stato deciso di registrare gli eventi di modifica di quest'ultimi da parte di utenti fittizi simulati tramite la definizione di alcuni task all'interno dello script Python. Inoltre, le operazioni svolte su ciascun prodotto sono state ripartite in modo che ognuno di essi avesse un peso percentuale diverso all'interno dell'intera collezione.

È stato possibile dunque assegnare dei pesi alle diverse operazioni di scrittura, grazie al *decorator* `@task` esportato da Locust, come mostrato in figura 5.2. Questo serve non solo ad identificare una funzione come un task, ma anche per assegnare un peso che ne specifichi il rapporto di esecuzione.

```
@task(1)
@tag("car")
def addHistoryForCarDescription(self):
    WriteTest(self.client).setCarSpeed()

@task(5)
@tag("truck")
def addHistoryForTruckWeight(self):
    WriteTest(self.client).setTruckWeight()

@task(10)
@tag("boat")
def addHistoryForBoatColor(self):
    WriteTest(self.client).setBoatColor()
```

Figura 5.2 Esempio di chiamata di un task. Si noti la differenza nei pesi di ciascun task. Il *decorator* `@tag` è stato utile nell'esecuzione dello script per includere o escludere alcuni tipi di task durante l'esecuzione, specificando il tag tramite il comando `locust -T [tag_name]`.

Per questi prodotti è stato scelto arbitrariamente un peso pari a 1 per il prodotto *car*, un peso pari a 5 per il prodotto *truck* e un peso pari a 10 per il prodotto *boat*. In questo modo, qualunque fosse la dimensione finale della collezione, i documenti corrispondenti alle scritture del prodotto *car* avrebbero costituito circa il 7% del totale, quelli corrispondenti a *truck* circa il 30% e quelli corrispondenti a *boat* circa il rimanente 63%. Così facendo, è stato possibile analizzare le prestazioni dei successivi test di lettura anche in base al numero di documenti da analizzare per ciascun prodotto.

In un ulteriore file, arbitrariamente denominato `writeTest.py`, sono stati definiti i task veri e propri che vengono esportati e cui si fa riferimento nel `locustfile.py`, come mostrato in figura 5.2. In ciascuna funzione viene definita la query da eseguire e le sue variabili, seguendo il modello già definito per l'applicazione tramite *schema* e *model* di Mongoose. Dopodiché viene eseguita la chiamata, come illustrato in figura 5.3.

```
def setCarSpeed(self):

    newValue = round(random.uniform(8.5, 10.5), 1)

    query = QUERY["setContent"]

    variables = {
        "nodeId": "IRT4gWPPSLGXUOwelmA19",
        "content": {
            "code" : "SPEED",
            "group" : "CAR_SPEC",
            "type" : "measure",
            "value" : newValue
        }
    }

    query['variables'] = merge(query['variables'], variables)

    response = self.client.post(
        "/graphql",
        name = "GraphQL - setCarSpeed",
        data = json.dumps(query),
        headers = header
    )
```

Figura 5.3 Esempio di definizione di task in scrittura. Viene definita la query (esportata da un altro file contenente la definizione delle query GraphQL interessate) e le variabili che definiscono il tipo di modifica in scrittura che viene eseguita. Infine, viene eseguita la query che, in questo esempio, corrisponde alla modifica da parte di un utente del *Content Type* `SPEED` appartenente al *Content Type Group* `CAR_SPEC` con un nuovo valore `newValue` casuale a ogni esecuzione del task, tra i valori 8.5 e 10.5.

5.2.1 Prestazione dei test in scrittura

Collegandosi alla porta 8089 dell'indirizzo locale dopo aver eseguito lo script e selezionato il numero di utenti in esecuzione da simulare, è possibile osservare statistiche e tempi di esecuzione dei vari task. In questo caso, tuttavia, si è trattato di dati poco significativi, anche nei casi più estremi (ossia con circa 100.000 documenti e 100 utenti attivi contemporaneamente) poiché le operazioni eseguite corrispondono a semplici aggiunte di un documento a una collezione di dati, operazioni che hanno richiesto sempre in media qualche decina di millisecondi.

5.3 Test in lettura

I test più significativi sono stati quelli in lettura, poiché si può facilmente immaginare che, con l'aumentare della dimensione della collezione di dati, aumenti anche il tempo richiesto per leggerne i documenti, anche se pochi.

Analogamente ai testi in scrittura, è stato definito un file `locustfile.py` in cui vengono chiamati i task (definiti in un ulteriore file `readTest.py`) che eseguono diversi tipi di letture per ciascun prodotto: alcuni ricercano tutti i documenti corrispondenti a quel prodotto, altri solo di un suo specifico *Content Type Group* o *Content Type*, diversificando in questo modo il numero di documenti da filtrare.

A differenza dei test in scrittura, non è stato specificato alcun peso di esecuzione per i diversi task, in modo che fossero eseguiti tutti con la stessa probabilità, permettendo di avere un confronto finale più significativo.

I task definiti nel file `readTest.py` seguono la stessa logica di quelli definiti per i test di scrittura, chiamando le opportune query e utilizzando le dovute variabili, come illustrato in figura 5.4. Per i test che implementano la paginazione è stato scelto di ritornare 50 documenti tra quelli che avrebbero avuto una corrispondenza con la query, mentre per i test applicati alla soluzione più semplice, senza paginazione, è stato scelto di ritornarne 10 per non sovraccaricare il sistema.

Per ciascuna soluzione (ossia la prima più semplice senza paginazione, la seconda che vede l'applicazione della paginazione tramite i metodi *skip* e *limit* e l'ultima, che applica la paginazione ricorrendo agli indici e alle *range queries*) sono stati eseguiti i test in lettura per ciascuna combinazione di casi, ossia considerando

```
def historyForCarNode(self):

    query = QUERY["getHistory"]

    variables = {
        "node": "IRT4gWPPSLGXUOwelmA19",
        "offset": current_time_millis(),
        "limit": 50,
        "direction" : 1
    }

    query['variables'] = merge(query['variables'], variables)

    response = self.client.post(
        "/graphql",
        name = "GraphQL - getHistoryForCarNode",
        data = json.dumps(query),
        headers = header
    )
```

Figura 5.4 Esempio di definizione di task in lettura. Questo task è stato utilizzato per la soluzione basata su paginazione con *range queries*, come si evince dal valore di `offset` corrispondente alla data attuale, usato come selettore per gli eventi precedenti a questo valore (dato il valore di `direction` pari a 1). Il numero di eventi ritornati è specificato da `limit`, così come precedentemente spiegato nel capitolo 4.

collezioni di 1.000, 10.000, 50.000 e 100.000 documenti, sia con 10 che con 100 utenti contemporaneamente attivi.

5.3.1 Test con collezione di 1.000 documenti

Il primo test è stato effettuato su una collezione di circa 1.000 documenti, corrispondenti ai tre tipi di prodotto nelle percentuali descritte nella sezione 5.2. Nelle figure seguenti sono riportati i tempi medi di risposta nei vari casi, distinguendo i task eseguiti per il prodotto *car* (a sinistra), per il prodotto *truck* (al centro) e per il prodotto *boat* (a destra). In questo modo si può notare bene la differenza nei tempi di esecuzione con l'aumentare della dimensione della collezione di documenti da ricercare.

Si può notare in figura 5.5 che, senza implementare la paginazione, con 100 utenti attivi per il prodotto *boat* già si raggiungono tempi di risposta discutibili, intorno ai 500 ms. Negli altri due casi, invece, i tempi sono più accettabili e pressoché identici, come illustrato in figura 5.8. Per quanto riguarda i tempi di risposta con 10 utenti attivi, in tutti e tre i casi si tratta di tempi più che accettabili.

Con questo primo test dunque si possono già vedere le potenzialità della paginazione, ma tutto sommato le tre soluzioni non differiscono molto tra di loro in termini di prestazioni e dunque si potrebbe decidere di scegliere quella che richiede meno manutenzione a livello di codice.

Figura 5.5 Tempi di risposta senza paginazione per 1.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

Figura 5.6 Tempi di risposta con paginazione a livello di query per 1.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

Figura 5.7 Tempi di risposta con paginazione tramite *range queries* per 1.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

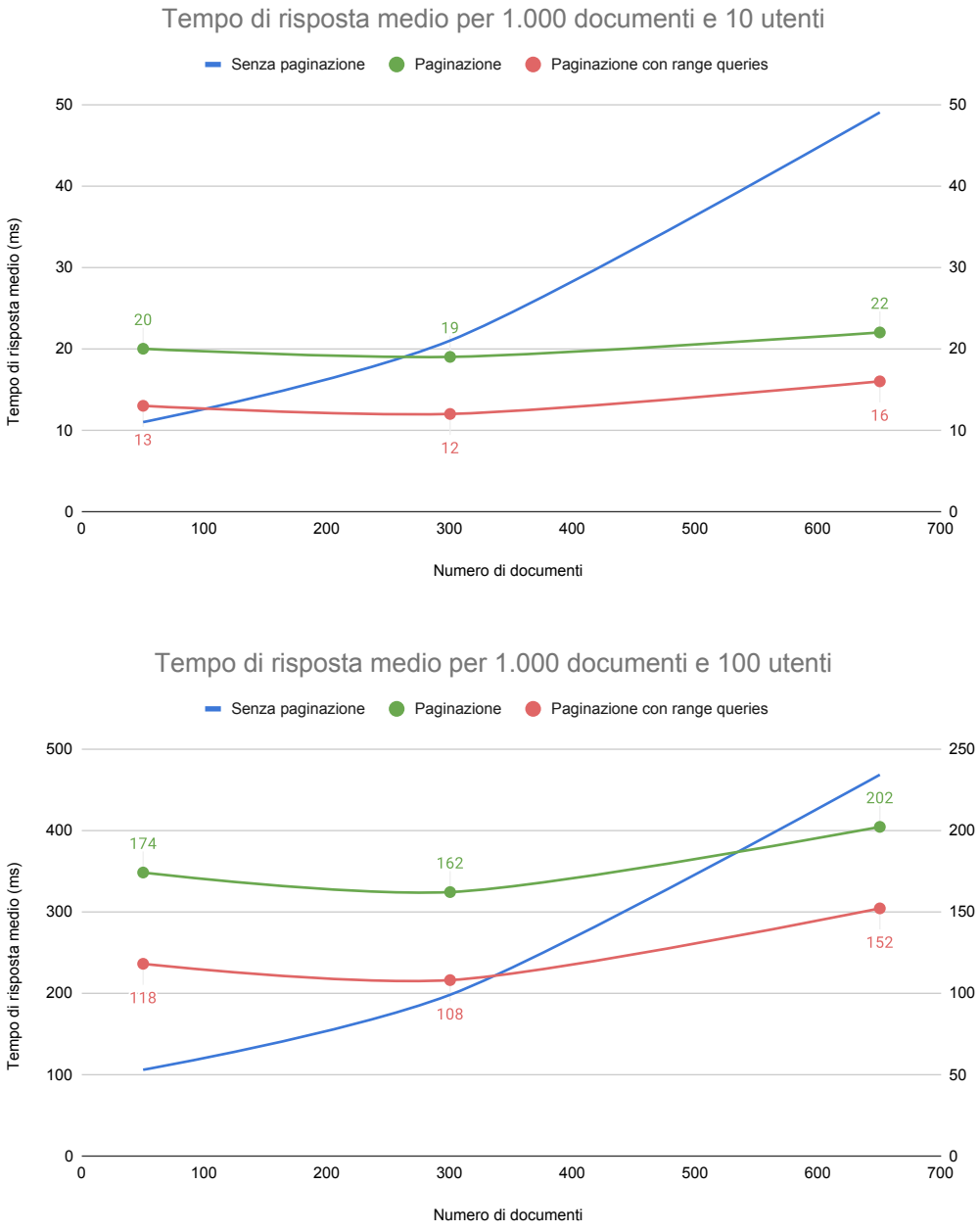


Figura 5.8 Confronto tra tempi di risposta medi nei tre diversi casi. I tempi per i test senza paginazione sono sempre riferiti alla scala a sinistra, poiché presenta tempi molto più grandi degli altri due casi, riferiti invece alla scala a destra. La soluzione senza paginazione sembra avere tempi più veloci per il prodotto *car*, ma è dovuto al fatto che i test senza paginazione sono stati effettuati ritornando solo 10 documenti, mentre negli altri casi 50.

5.3.2 Test con collezione di 10.000 documenti

Il secondo test è stato effettuato su una collezione di circa 10.000 documenti. Con questo secondo test, la soluzione senza paginazione inizia a vacillare anche con soli 10 utenti contemporaneamente attivi mentre le altre due soluzioni riescono invece a mantenersi basse e a valori pressoché identici.

In figura 5.12 si può notare molto bene però come la soluzione che implementa la paginazione tramite *range queries* si discosti molto (soprattutto nel caso di 100 utenti attivi) dall'implementazione della paginazione più semplice, con tempi di risposta medi pari circa alla metà.

Questo secondo test dimostra l'inefficacia di una soluzione che non adotta la paginazione, che al crescere della dimensione della collezione fornisce tempi di risposta non accettabili. Ciò nonostante, a riprova della sua inefficacia, questa soluzione è stata comunque testata anche con collezioni più ampie. Le soluzioni che adottano la paginazione, invece, sono più performanti e possono essere considerate entrambe valide, nonostante la paginazione più semplice restituisca dei tempi circa doppi.

Figura 5.9 Tempi di risposta senza paginazione per 10.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

Figura 5.10 Tempi di risposta con paginazione a livello di query per 10.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

Figura 5.11 Tempi di risposta con paginazione tramite *range queries* per 10.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

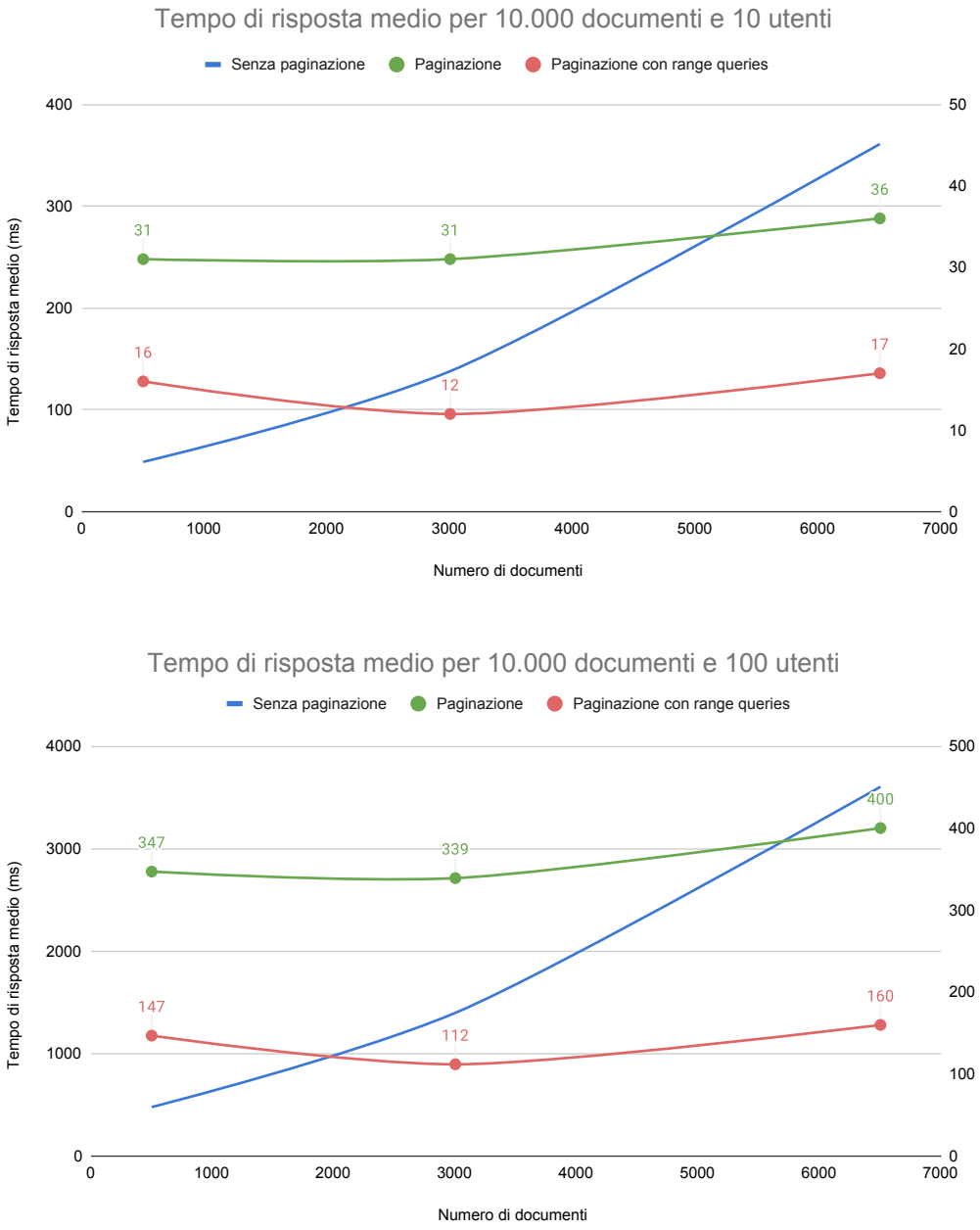


Figura 5.12 Confronto tra tempi di risposta medi nei tre diversi casi. I tempi per i test senza paginazione sono sempre riferiti alla scala a sinistra, poiché presenta tempi molto più grandi degli altri due casi, riferiti invece alla scala a destra. In questo caso, infatti, la soluzione senza paginazione ha tempi circa 10 volte maggiori rispetto alle altre due.

5.3.3 Test con collezione di 50.000 documenti

Con questo terzo test, oltre a dimostrare ulteriormente la deriva presa dalla soluzione senza paginazione (chiaramente non applicabile per una collezione di dati così ampia, in quanto raggiunge tempi di risposta di circa 25 secondi con 100 utenti attivi come mostrato in figura 5.13), si nota in particolare l'effettivo vantaggio nell'effettuare una ricerca all'interno di una collezione di MongoDB sfruttando gli indici che, nel caso più estremo di utenti contemporaneamente attivi, ha tempi di risposta circa 10 volte più bassi e lo si può notare bene in figura 5.16. Questo risultato ci permette già di escludere la semplice paginazione come soluzione ottimale, a favore di quella implementata con le *range queries*.

È stato dunque effettuato un ultimo test, con quello che è stato considerato dall'azienda un realistico caso estremo da sottoporre a stress, per notare il comportamento della soluzione basata su paginazione con *range queries*.

Figura 5.13 Tempi di risposta senza paginazione per 50.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

Figura 5.14 Tempi di risposta con paginazione a livello di query per 50.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

Figura 5.15 Tempi di risposta con paginazione tramite *range queries* per 50.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

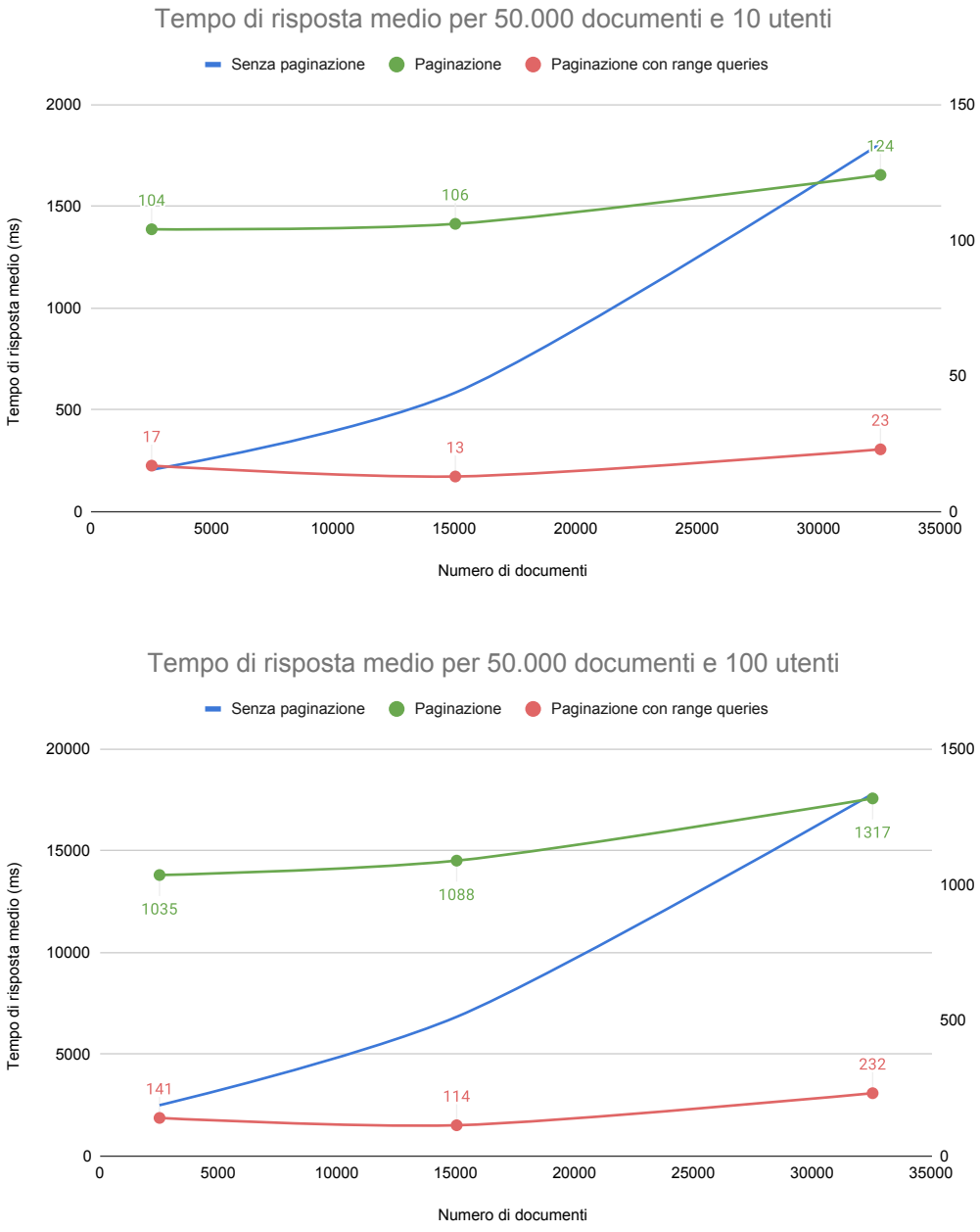


Figura 5.16 Confronto tra tempi di risposta medi nei tre diversi casi. I tempi per i test senza paginazione sono sempre riferiti alla scala a sinistra, poiché presenta tempi molto più grandi degli altri due casi, riferiti invece alla scala a destra. Anche in questo caso, infatti, la soluzione senza paginazione ha tempi circa 10 volte maggiori rispetto alle altre due.

5.3.4 Test con collezione di 100.000 documenti

Quest'ultimo definitivo test sancisce la soluzione con paginazione tramite *range queries* come soluzione più performante, che fornisce tempi di risposta ottimi anche nel caso limite di 100.000 documenti e 100 utenti contemporaneamente attivi, restituendo i dati in meno di 350 ms (in media), senza discostarsi troppo dai tempi registrati per 50.000 documenti.

Non altrettanto bene hanno performato le altre due soluzioni che, con 100 utenti attivi, hanno registrato tempi ben superiori (circa 2 secondi per la paginazione semplice e oltre i 40 secondi per la soluzione senza paginazione) e, nel tentativo di eseguire i task relativi al prodotto *boat* che, come descritto nella sezione 5.2, costituisce circa il 63% dei documenti (quindi in questo caso circa 63.000 documenti), hanno bloccato l'applicazione, interrompendo le chiamate (motivo per cui in figura 5.21 per queste due soluzioni non è presente il tempo per il prodotto *boat*).

Un miglior tempo di risposta implica la possibilità di gestire un maggior numero di richieste nello stesso lasso di tempo, aumentando di conseguenza il numero di richieste al secondo, ossia il numero di task eseguite al secondo. Come si può ben vedere in figura 5.17, non vi è assolutamente paragone tra le tre soluzioni analizzate: la paginazione con *range queries* permette di avere un'esecuzione dell'applicazione performante anche se sottoposta a stress, senza grosse difficoltà nel gestire ampie collezioni di dati, differentemente dalle altre due soluzioni che hanno portato a un blocco dell'applicazione.

Senza paginazione

	10 utenti			100 utenti		
	car	truck	boat	car	truck	boat
Numero di richieste	472	132	45	425	92	
Tempo medio di risposta (ms)	440	1457	4073	4566	7291	
Richieste al secondo	22.8	7.1	2.5	19.6	3.9	

Paginazione semplice

	10 utenti			100 utenti		
	car	truck	boat	car	truck	boat
Numero di richieste	1510	1328	1120	1333	1108	
Tempo medio di risposta (ms)	144	167	210	1578	1795	
Richieste al secondo	70.3	59.4	46.4	63.6	53.9	

Paginazione con range queries

	10 utenti			100 utenti		
	car	truck	boat	car	truck	boat
Numero di richieste	13067	15246	6642	13394	17544	6595
Tempo medio di risposta (ms)	17	13	32	147	120	328
Richieste al secondo	641.7	753.4	309.8	662	920.8	316.7

Figura 5.17 Confronto tra le principali statistiche misurate nei tre differenti casi. La soluzione con paginazione tramite *range queries*, oltre a garantire tempi di risposta medi migliori, permette di gestire una maggiore quantità di richieste, fornendo un'ottima esperienza utente anche in situazioni limite con una collezione di 100.000 documenti e 100 utenti contemporaneamente attivi.

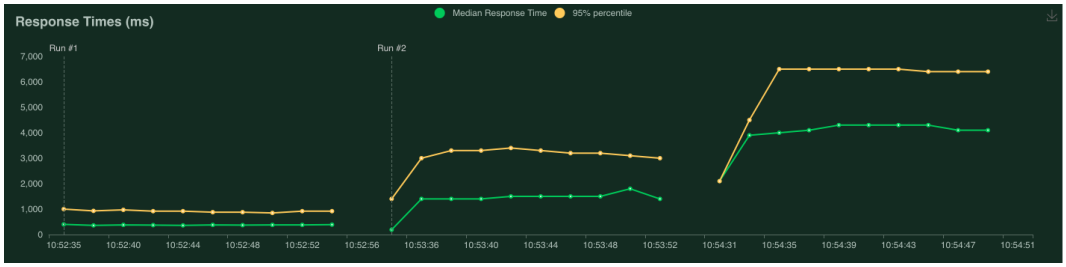


Figura 5.18 Tempi di risposta senza paginazione per 100.000 documenti. In alto con 10 utenti attivi. I tempi di risposta per 100 utenti attivi non sono stati registrati, in quanto all'esecuzione dello script, l'applicazione non è riuscita a gestire il sovraccarico di richieste unito alla dimensione del risultato da restituire.

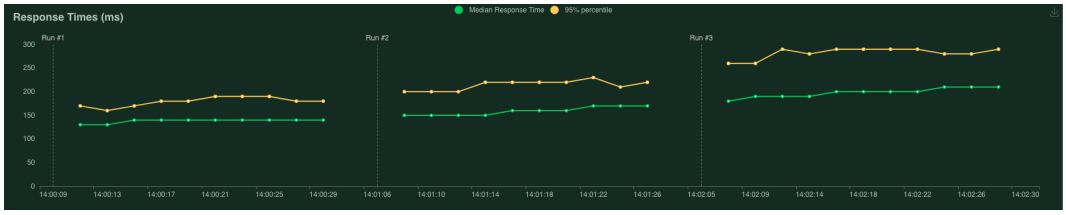


Figura 5.19 Tempi di risposta con paginazione a livello di query per 100.000 documenti. In alto con 10 utenti attivi. I tempi di risposta per 100 utenti attivi non sono stati registrati, in quanto all'esecuzione dello script, l'applicazione non è riuscita a gestire il sovraccarico di richieste unito alla dimensione del risultato da restituire.

Figura 5.20 Tempi di risposta con paginazione tramite *range queries* per 100.000 documenti. In alto con 10 utenti attivi, in basso con 100 utenti attivi.

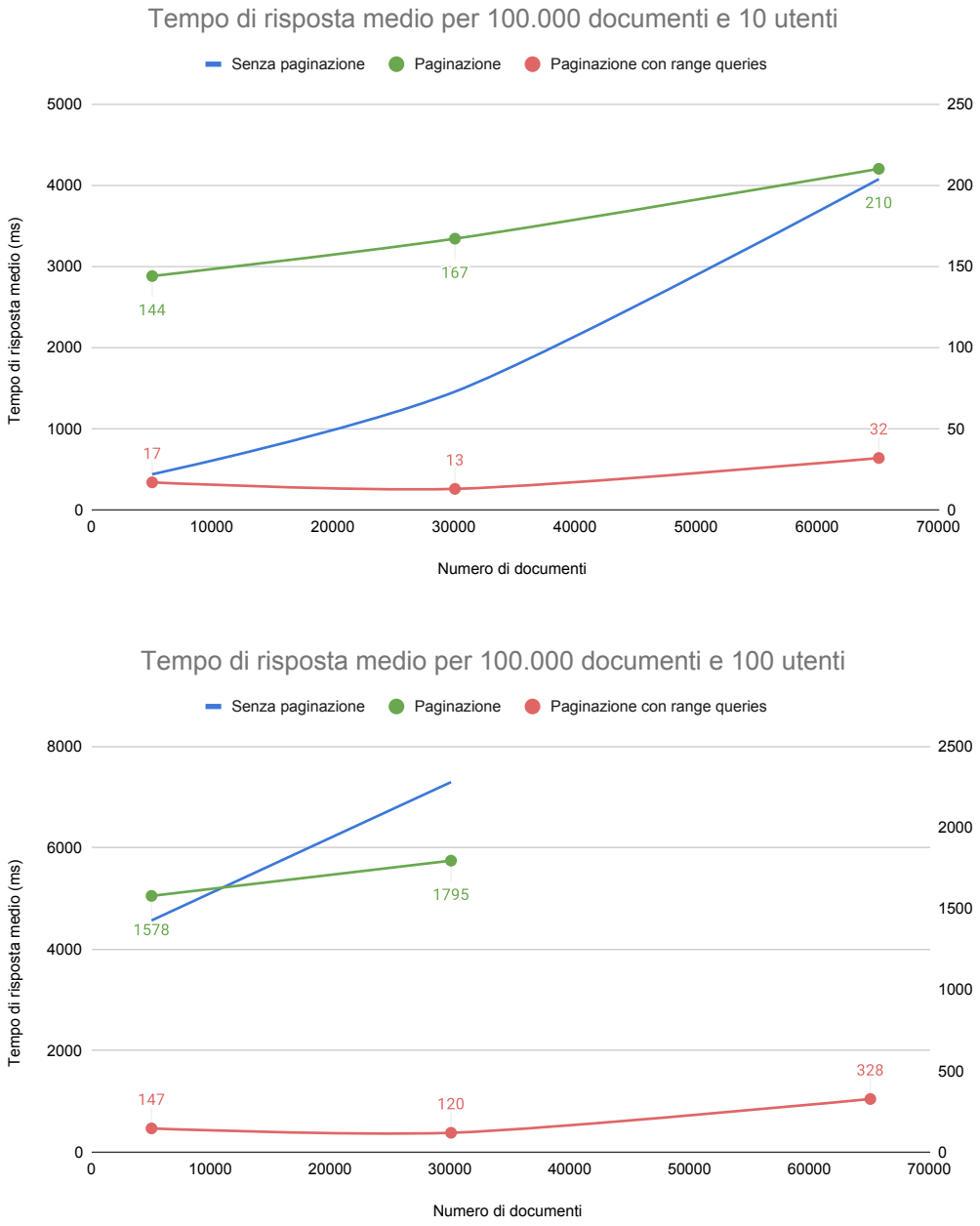


Figura 5.21 Confronto tra tempi di risposta medi nei tre diversi casi. I tempi per i test senza paginazione sono sempre riferiti alla scala a sinistra, poiché presenta tempi molto più grandi degli altri due casi, riferiti invece alla scala a destra. In quest'ultimo test è evidente il dato mancante con 100 utenti attivi per il prodotto *boat* per le soluzioni senza paginazione e con paginazione semplice, dovuto all'applicazione che si è bloccata a fronte di numerose richieste al secondo di un'ampia collezione di dati.

Capitolo 6

Event sourcing pattern e CQRS *pattern*

6.1 Le proiezioni

Nell'*event sourcing*, le proiezioni (note anche come *view model* o *query model*) forniscono una vista del sottostante *data model* basato sugli eventi. Spesso rappresentano la logica di traduzione dal *write model* sorgente al *read model* e vengono usati sia nei *read model* che nei *write model*.

6.1.1 Proiezioni nel *read model*

Uno scenario comune è utilizzare gli eventi creati nel *write model* e calcolare una vista per il *read model*. Questo oggetto può essere poi memorizzato in un database differente per essere poi utilizzato dalle *query*.

Un insieme di eventi può anche essere considerato un punto di partenza per generare un altro insieme di eventi. In questo caso la proiezione prende il nome di trasformazione.

6.1.2 Proiezioni nel *write model*

Un'altra forma di proiezione è chiamata *stream aggregation*. Si tratta di un processo di costruzione dello stato attuale del *write model* a partire dal flusso di eventi. Durante l'aggregazione, gli eventi sono applicati uno ad uno in ordine di apparizione. Lo

scopo principale della *stream aggregation* è quello di ricostruire lo stato attuale per validare un comando che viene eseguito su di esso, come illustrato nello schema in figura 6.1.

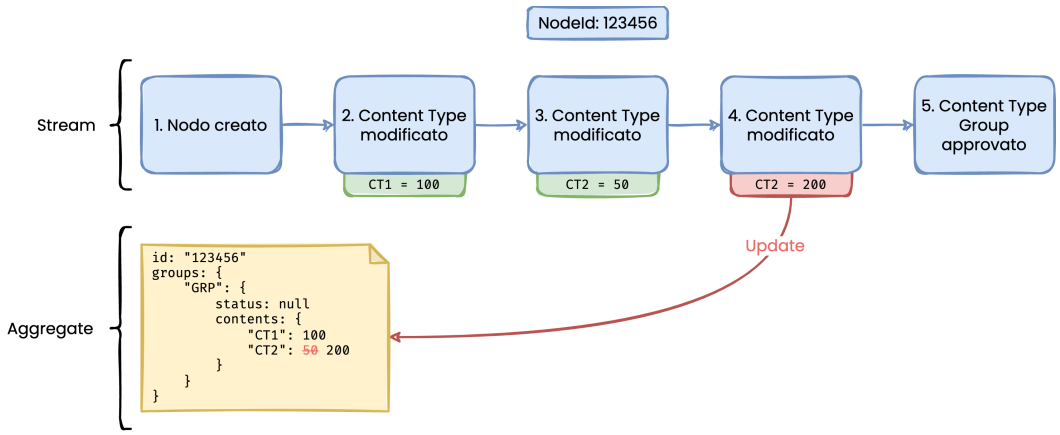


Figura 6.1 Esempio di aggregato costruito a partire dall'applicazione dei singoli eventi. Lo stato finale avrà corrispondenze con gli eventi che più recentemente hanno modificato un'entità.

Le proiezioni dovrebbero essere considerate temporanee ed usa e getta. Questo è proprio uno dei loro vantaggi chiave, ossia poter essere distrutte, reinventate e ricreate a piacimento; per questo motivo non dovrebbero essere considerate come fonte di verità (a differenza dei semplici eventi).

Esistono alcuni incroci concettuali tra la proiezione e il *read model*, il che può creare confusione. Basti pensare che un *read model* è costituito da molteplici proiezioni e che quindi le proiezioni sono un metodo di popolazione del *read model* e corrispondono a parti discrete dell'intero *read model*.

Ricorrendo alla *stream aggregation*, viene ricreato lo stato corrente dell'entità su cui poi poter mettere in atto eventuali validazioni e proseguire col flusso del programma. Risulta fin da subito chiaro però che, in presenza di molte entità da ricostruire attraverso numerosi eventi memorizzati, lo *stream aggregation* diventa un processo dispendioso e poco performante [5].

6.2 Il CQRS *pattern* applicato all'*event sourcing*

Per ovviare a quest'ultimo problema, è comune ricorrere all'utilizzo di un altro *pattern*, spesso proprio utilizzato assieme all'*event sourcing*, ossia il CQRS.

Il CQRS è stato definito da Greg Young, che lo ha descritto come:

> La semplice creazione di due oggetti dove precedentemente ne era presente solamente uno. La separazione occorre a seconda che i metodi corrispondano a un *command* o a una *query* (la stessa definizione usata da Meyer in *Command and Query Separation*: un *command* è un qualunque metodo che modifichi lo stato e una *query* è un qualunque metodo che ritorni un valore).

Utilizzando il CQRS, si dovrebbe avere una separazione esatta tra il *write model* e il *read model*. Questi due *model* dovrebbero essere processati da oggetti distinti e non essere concettualmente legati tra loro. Questi oggetti non sono strutture di memorizzazione fisiche ma sono, ad esempio, *command handler* e *query handler*. Non sono dunque collegati a dove e come i dati sono memorizzati, bensì solo connessi a come vengono elaborati. I *command handler* sono responsabili della gestione dei comandi, del cambiamento dello stato o di altri effetti collaterali. I *query handler* sono invece responsabili della restituzione del risultato per la *query* richiesta [5].

Il concetto cardine del CQRS, ossia *Command Query Responsibility Segregation*, è dunque quello di poter utilizzare un modello differente per aggiornare le informazioni in un database, rispetto a quello che viene utilizzato per leggerle.

Il CQRS permette di suddividere quell'unico modello concettuale (valido sia per scrittura che per lettura, come ad esempio quello proposto dall'*event sourcing pattern*) in modelli separati a seconda dell'operazione da svolgere, per l'appunto *Command* (in caso di aggiornamento dei dati) e *Query* (in caso di lettura), come si vede in figura 6.2.

Nella sua applicazione con l'*event sourcing*, tramite l'utilizzo dei cosiddetti *event handler* (ossia opportune funzioni che si mettono in ascolto di cambiamenti in scrittura su un particolare database) permette di aggiornare, man mano che vengono registrati degli eventi, uno stato che verrà poi utilizzato in fase di lettura come dato già pronto, validato e aggiornato per essere visualizzato.

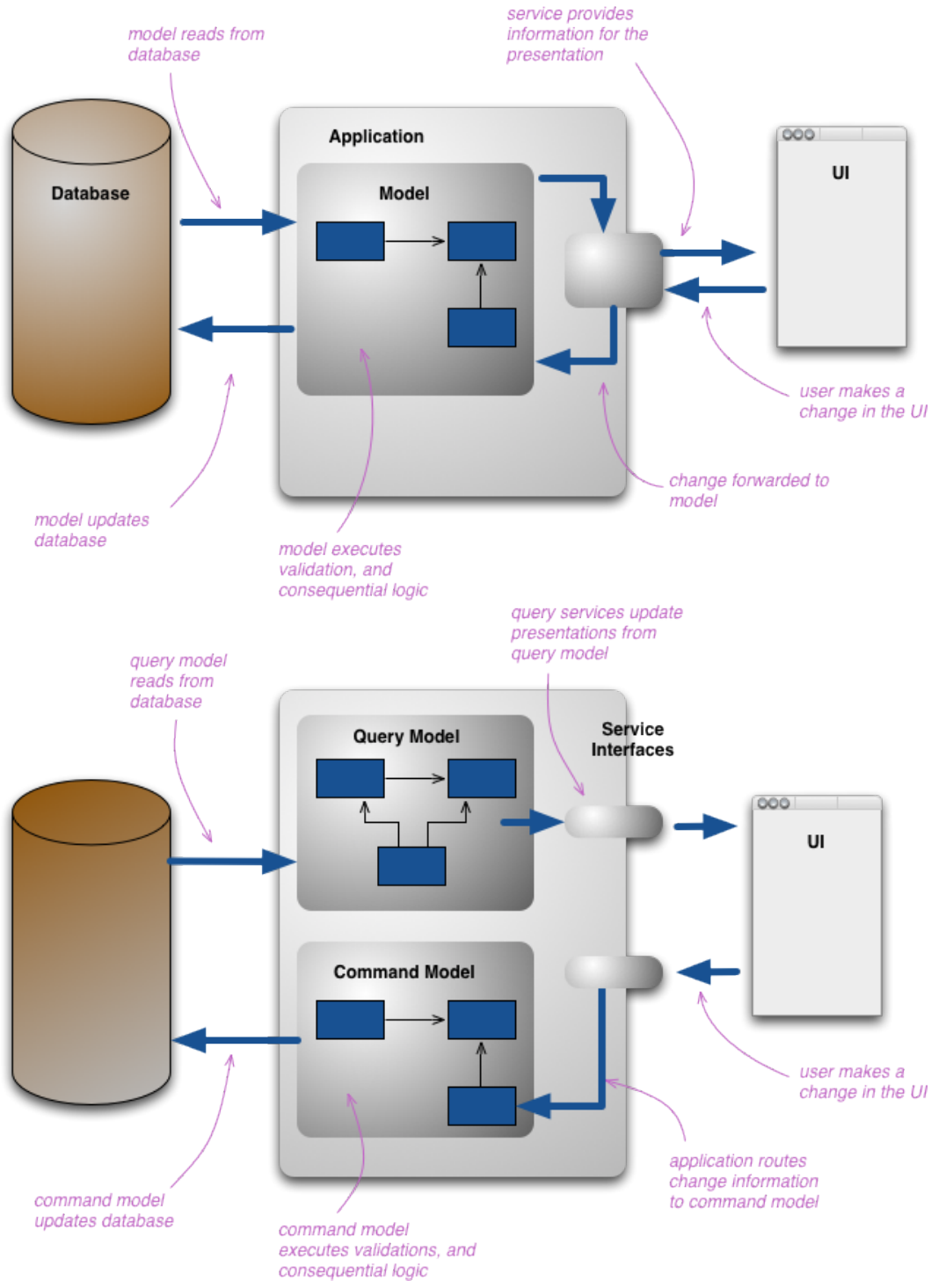


Figura 6.2 In queste immagini è visibile la differenza tra l'utilizzo di un singolo *model* per lettura e scrittura, e la soluzione proposta dal CQRS *pattern*, che differenzia *read model* e *write model*, chiamati *Query Model* e *Command Model* rispettivamente [1].

6.3 Realizzazione di un *event handler* per calcolo di statistiche

Come accennato alla fine del Capitolo 2, nella sezione *Dashboard* sono visualizzate alcune statistiche, riguardanti tutte le gerarchie e i nodi creati nella sezione *Hierarchies*. Queste statistiche mutano in seguito alle operazioni effettuate dagli utenti nella sezione *Hierarchies*, siano esse modifiche dei contenuti, approvazioni di *Content Type Group*, creazioni di nuovi nodi o gerarchie, eliminazione di nodi o modifiche di parentela all'interno di una gerarchia. Questi sono tutti eventi che, come descritto in precedenza, è stato deciso di registrare nell'*event store*, dunque è naturale pensare di poter sfruttare questi stessi eventi per calcolare queste statistiche non appena viene registrato un nuovo evento.

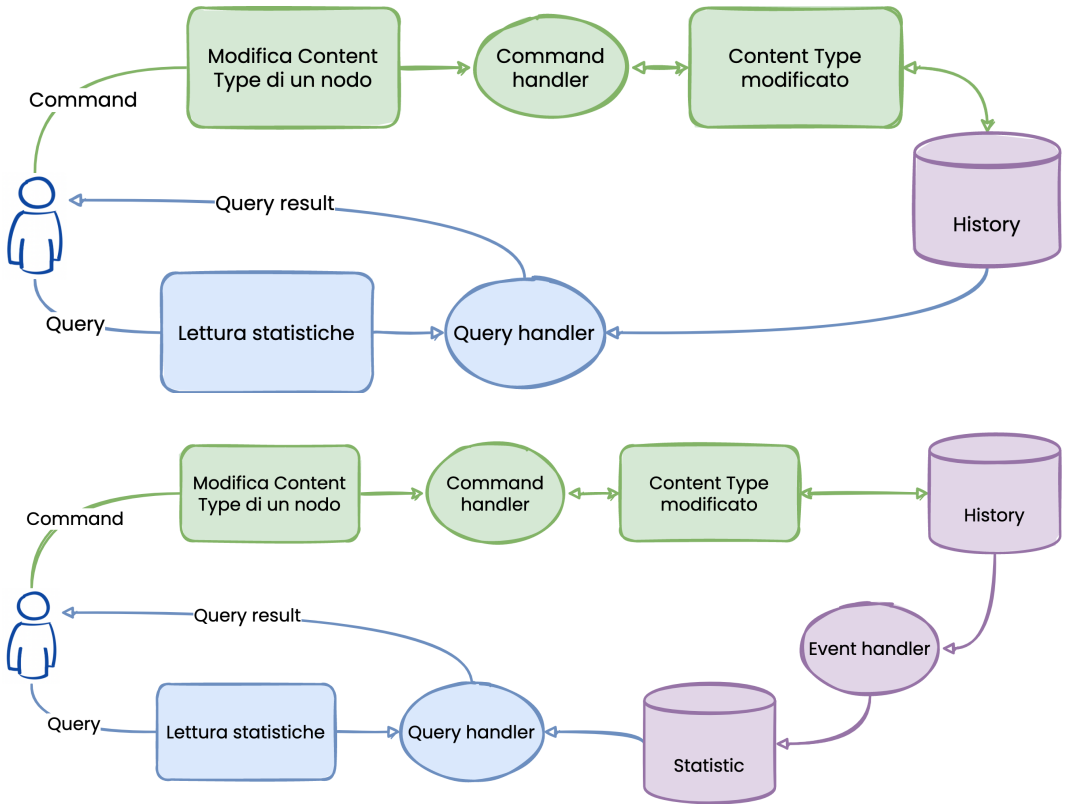


Figura 6.3 In alto è riportato lo schema concettuale della più semplice realizzazione di *write* e *read model* secondo il CQRS *pattern*, ossia utilizzando un unico database. In basso è riportato lo schema di come è stato invece realizzato: tramite la creazione di un nuovo database *Statistic* aggiornato utilizzando un *event handler* in ascolto degli eventi che vengono aggiunti nel database *History*.

6.3.1 Apertura di un *change stream* in attesa di un evento

In un nuovo file, utilizzato successivamente come sorgente, è stata definita tutta la logica per elaborare gli eventi, ridurli a un aggregato e calcolare le statistiche interessate.

In particolare, è stato utilizzato il metodo `watch` esposto da MongoDB e utilizzabile su una collezione (*History* in questo caso) recuperata collegandosi necessariamente a un *replica set* del database vero e proprio. Un *replica set*, in MongoDB, è un gruppo di processi `mongod` (i principali processi *daemon* del sistema MongoDB) che mantengono lo stesso *data set*. Forniscono ridondanza e alta disponibilità e sono alla base di ogni distribuzione di produzione [13] [14].

Questo metodo *watch* apre un cosiddetto *change stream cursor* sulla collezione. In questo modo è possibile sottoscrivere a tutti i cambiamenti che avvengono in una singola collezione e reagire immediatamente a essi [15].

In questo caso un cambiamento corrisponde all'inserimento di un nuovo evento nella collezione *History*. Dunque, non appena viene intercettato un evento, questo viene utilizzato per creare un aggregato. Questo concetto segue quanto rappresentato nello schema in basso in figura 6.3.

6.3.2 Creazione di un aggregato

Una volta catturato l'evento da inserire nell'*event store*, si utilizzano le informazioni contenute in esso per ricavare tutti gli eventi precedenti riferiti allo stesso nodo e, applicandoli in modo consecutivo, si ottiene un aggregato corrispondente allo stato attuale in cui si trova il nodo in questione. A tale scopo, ho creato un nuovo tipo `EventNode` così definito:

```
type EventNode = {
    id: NodeId;
    parent: NodeId | null;
    groups: GroupInfo;
    status: Status;
    published: boolean;
};
```

definendo a loro volta `Status` e `GroupInfo` come segue:

```
enum Status {
    NEW = 'new',
    DRAFT = 'draft',
    PUBLISHED = 'published',
    REMOVED = 'removed',
    APPROVED = 'approved'
}

type GroupInfo = {
    [key: GroupCode]: {
        status: 'approved' | null;
        contents: {
            [key: ContentTypeCode]: {
                required: boolean;
                value: Content | null;
            };
        };
    };
};
```

Nel suo *Domain Driven Design* (anche conosciuto come *the blue book*), Eric Evans discute della natura granulare degli oggetti e fornisce una definizione di aggregato:

> Un aggregato è un gruppo di oggetti associati che trattiamo come unità ai fini della modifica dei dati. Ogni aggregato ha una radice e dei limiti. I limiti definiscono cosa vi è all'interno dell'aggregato, mentre la radice corrisponde a una singola entità specifica contenuta nell'aggregato. La radice è l'unico membro dell'aggregato che gli oggetti esterni possono referenziare, mentre oggetti all'interno dei limiti possono referenziarsi a vicenda. Le entità al di fuori della radice hanno un'identità locale che deve essere distinguibile solo all'interno dell'aggregato, poiché nessun oggetto esterno può vederle al di fuori del contesto dell'entità radice [16].

Seguendo questa definizione, in questo caso, `EventNode` corrisponde al *domain object*, ma è anche costituito da diversi oggetti, come i *Content Type Group*, lo stato del nodo, l'`Id` del nodo padre, ecc. L'`EventNode` è dunque l'aggregato e ogni parte di dato necessaria per formare l'`EventNode` è un'entità. La radice deve essere una singola entità specifica dell'aggregato, in questo caso l'`Id` del nodo.

Nello specifico, le entità di `EventNode` sono:

- Il GUID del nodo
- Il GUID del nodo padre; se non è presente, il nodo in questione è un nodo radice (o gerarchia)
- La lista di tutti i *Content Type Group*, per cui sono definiti:
  - Il suo stato, ossia se sia attualmente approvato
  - La lista dei suoi contenuti, definendo per ciascuno se siano campi obbligatori (`required`) e il loro valore attuale
- Lo stato attuale del nodo, fondamentale per calcolare le statistiche richieste
- Un valore che indica se il nodo abbia avuto almeno una volta tutti i suoi *Content Type Group* contemporaneamente attivati

Durante la creazione dell'aggregato, se viene processato un evento corrispondente alla creazione del nodo, a un cambio di parentela o alla sua rimozione, vengono opportunamente aggiornati i valori corrispondenti al numero totale di nodi, di nodi foglia, di nodi gerarchici e di nodi pubblicati.

Una volta utlimata la creazione dell'aggregato, invece, vengono aggiornate le statistiche riguardanti il numero di nodi bozza, di nodi nuovi e di quelli non completati.

Tutte le statistiche così calcolate corrispondono però a un dato parziale che viene quindi sommato al valore di un documento già presente in una nuova collezione, chiamata *Statistic*. In questo modo, dalla sezione *Dashboard*, interrogando la collezione *Statistic* si possono ottenere i dati più recentemente elaborati e visualizzarli.

6.3.3 Svantaggi e possibili soluzioni

Se la creazione di un aggregato scatenata dalla registrazione di un nuovo evento permette di avere una corrispondenza precisa con quello che è l'ultimo stato noto di un nodo, non è detto che il dato visualizzato nella sezione *Dashboard* sia a sua volta aggiornato, poiché potrebbe essere stato ottenuto dalla collezione *Statistic* in un momento in cui l'*event handler* (che agisce parallelamente al resto dell'applicazione) non abbia ancora sovrascritto il dato.

Inoltre, la cattura di un nuovo evento da parte dell'*event handler* fa si che si debba ricreare l'aggregato, processando nuovamente tutti gli eventi corrispondenti al nodo in questione.

Memorizzare l'aggregato precedente per poi applicare su di esso un nuovo evento entrante permette di evitare di ri-processare tutti gli eventi, ma richiede memoria per ogni aggregato di ogni nodo cui corrisponde almeno un evento, traducendosi in un progressivo rallentamento di tutta l'applicazione.

Si potrebbe inoltre facilmente assumere di creare un unico mega aggregato. Tuttavia, ciò causerebbe più problemi nel lungo termine. Aggregati più piccoli e concisi, focalizzati su un unico aspetto permettono di avere un sistema generalmente più efficiente, preservando il *business context* dove necessario [5].

La soluzione migliore sarebbe quella di schedulare una sostituzione all'interno della collezione *History* di un certo numero di eventi con un unico evento corrispondente al loro aggregato, permettendo in questo modo di tenere sotto controllo le dimensioni della collezione e di dover successivamente processare meno eventi per crearne uno nuovo. Questo implica però la perdita delle informazioni riguardanti gli eventi più datati.

Le soluzioni dunque sono molteplici e nessuna permette di avere dei vantaggi senza alcun tipo di compromesso. Nel caso specifico del CoolPIM, però, è stato ritenuto preferibile e accettabile il ri-processare tutti gli eventi per ricostruire un aggregato valido, conservando dunque tutti gli eventi relativi a un singolo nodo.

Capitolo 7

Conclusioni

Quest'attività formativa presso l'azienda Coolshop mi ha permesso di inserirmi in un progetto già avviato, seguendo delle regole di scrittura del codice già consolidate dall'azienda, contribuendo attivamente all'introduzione di nuove funzionalità, studiandone prestazioni e motivazioni, approfondendo aspetti, tecnologie e linguaggi a me già conosciuti (come TypeScript, React, MongoDB) e scoprendone di nuovi.

Tra questi, ho avuto modo di conoscere e approfondire l'*event sourcing pattern* e tutte le sue possibili declinazioni all'interno di un'applicazione e come questo si combini con altri *pattern* per fornire un servizio migliore. Approfondendo questo concetto, mi è stato possibile capirne bene il funzionamento, tutte le potenzialità derivanti dalla sua applicazione, per quali scopi fosse possibile utilizzarlo e se fosse conveniente farlo, soprattutto all'interno del contesto del software aziendale CoolPIM.

In questo caso, infatti, si è rivelato essere la soluzione migliore per fornire un servizio di *auditing*, ed è stato fondamentale studiarne le prestazioni, scoprendo nuove strategie e soluzioni per fornire un servizio adeguato e performante anche nei casi più estremi. Fondamentale è stato lo studio della paginazione, senza la cui applicazione risulterebbe impossibile realizzare un *read model* accettabile. Ciò ha messo in risalto l'importanza degli eventi, in particolare per ciò che riguarda la loro immutabilità e la loro verità intrinseca. Per questo motivo si è ritenuto necessario e preferibile conservare tutti gli eventi, ricreando quando necessario degli aggregati, a scapito di operazioni più lente e non sempre immediatamente sincronizzate con il reale stato dei nodi: si è preferito dunque avere sempre a portata

di mano tutti gli eventi piuttosto che fornire un dato (corrispondente a semplici statistiche) immediatamente corretto, alleggerendo in ogni caso l'elaborazione in fase di esecuzione.

Ho potuto dunque concludere che, per i servizi implementati, ossia l'*auditing* e la creazione di un *read model* basato sugli eventi, l'utilizzo dell'*event sourcing pattern* si sia rivelata una scelta vincente, al netto della sua applicazione nel contesto del CoolPIM, per cui è stata prevista una dimensione massima della collezione degli eventi non superiore ai 100.000 documenti richiesti da al più 100 utenti attivi contemporaneamente; una realtà ben diversa da quella di molti altri applicativi in cui il numero di documenti è dell'ordine dei milioni. In un caso simile, probabilmente l'*event sourcing* non sarebbe stata la scelta migliore o avrebbe comunque richiesto un ulteriore lavoro attorno alla sua implementazione per rimpicciolire il *data set* associato ad esso, adottando altri *pattern* che cooperassero con esso, come già visto con il CQRS *pattern*.

Ringraziamenti

Il mio corso di studi è stato un lungo viaggio, vissuto guardando quasi mai alla destinazione. Non sono mancati gli imprevisti e le sorprese, ma soprattutto le persone, che hanno reso speciale questo viaggio.

Innanzitutto la mia famiglia, le persone a me più vicine, che hanno sempre sostenuto le mie scelte e il mio percorso. I miei amici, quelli che conosco da sempre, che hanno contribuito a formarmi come persona, e quelli che ho conosciuto durante gli studi, con cui ho condiviso questo percorso, soprattutto la fine, e di cui conservo ricordi meravigliosi e indimenticabili. Ringrazio Angelica, per aver sempre creduto in me e per essere sempre stata con me, anche quando eravamo distanti. Infine, ringrazio di aver avuto la fortuna di essere nato in un luogo, in un'epoca e in un contesto che mi hanno permesso di studiare e coltivare delle passioni, accrescendo la mia cultura, il mio pensiero critico e le mie capacità fino a questo giorno, in cui tutte queste esperienze confluiscono.

Questo è sicuramente un punto di arrivo ma, soprattutto, un punto di partenza per un nuovo lungo viaggio, accompagnato da queste stesse persone e da quelle future che conoscerò, cercando sempre di vivere le esperienze, cogliendo l'attimo, senza pensare alla destinazione.

Bibliografia

[1] Martin Fowler. CQRS. URL https://martinfowler.com/bliki/CQRS.html.

[2] Solutions. URL https://www.coolshop.it/en/solutions.

[3] Coolshop Srl. *Documentazione fornita dall'azienda*.

[4] EdPrice-MSFT. Modello di origine eventi - azure architecture center. URL https://docs.microsoft.com/it-it/azure/architecture/patterns/event-sourcing.

[5] Event Store Ltd. Beginner's guide to event sourcing | event store. URL https://www.eventstore.com/event-sourcing.

[6] Mongoose v6.5.2: Schemas. URL https://mongoosejs.com/docs/guide.html.

[7] Queries. URL https://www.apollographql.com/docs/react/data/queries/.

[8] Understanding pagination: REST, GraphQL, and relay. URL https://www.apollographql.com/blog/graphql/pagination/understanding-pagination-rest-graphql-and-relay/.

[9] cursor.skip() — MongoDB manual. URL https://www.mongodb.com/docs/manual/reference/method/cursor.skip/.

[10] Pagination. URL https://www.apollographql.com/docs/react/v2/data/pagination/.

[11] Indexes — MongoDB manual. URL https://www.mongodb.com/docs/manual/indexes/#std-label-indexes.

[12] Locust documentation — locust 2.12.0 documentation. URL https://docs.locust.io/en/stable/index.html.

[13] db.collection.watch() — MongoDB manual. URL https://www.mongodb.com/docs/manual/reference/method/db.collection.watch/.

[14] Replication — MongoDB manual. URL https://www.mongodb.com/docs/manual/replication/.

[15] Change streams — MongoDB manual. URL https://www.mongodb.com/docs/manual/changeStreams/#std-label-changeStreams.

[16] Eric Evans. *Domain Driven Design*.