POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Analisi ed implementazione di un'applicazione orientata agli eventi

Relatore:

Prof. Paolo GARZA

Candidato:

Arianna GENTILE

Luglio 2022

Sommario

L'Event Sourcing è un pattern architetturale sempre più utilizzato nelle moderne applicazioni Web orientate ai microservizi. Questo tipo di pattern ha suscitato grande interesse negli ultimi anni poiché promette il miglioramento di scalabilità, integrazione e tracciabilità per gli applicativi che ne fanno uso, cambiando il modo in cui i dati vengono gestiti nei sistemi distribuiti.

Questa tesi presenta un caso di studio sul modello di progettazione Event Sourcing e sui principi di un altro modello di progettazione, Command Query Responsibility Segregation, anche detto CQRS, strettamente associato al precedente. Inoltre verranno presentati i concetti di 'Domain-Driven-Design', il cui principale obiettivo è il velocizzare e semplificare lo sviluppo di applicazioni complesse, ed 'Event Store', database per la persistenza degli eventi, con le sue possibili implementazioni.

Lo scopo di questo elaborato è la presentazione di questi modelli e la loro applicazione all'interno di un reale progetto in linguaggio Java per poterne descrivere vantaggi e svantaggi relativi al loro utilizzo rispetto ad architetture tradizionali che si trovano tutt'oggi nella maggior parte degli applicativi.

Ringraziamenti

Indice

Elenco delle tabelle

Elenco delle figure

Acronimi

ES

Event Sourcing

CQRS

Command Query Responsibility Segregation

DDD

Domain Driven Design

RDBMS

Relational Database Management System

JMT

Java Modelling Tools

Capitolo 1

Introduzione

Al giorno d'oggi, la maggior parte delle aziende predilige lo spostamento dei propri applicativi sul cloud, poiché fornisce un utilizzo più efficiente delle risorse, una manutenzione dell'infrastruttura più semplice e costi inferiori, essendo dovuti nella maggior parte dei casi all'utilizzo di hardware standard.

Di questi tempi le grandi aziende che supportano il cloud producono costantemente nuovi concetti ed idee per la creazione di applicazioni distribuite. E' d'altra parte vero, però, che tutte queste nuove soluzioni non sono tutt'oggi ben consolidate.

Negli anni '70 sono state introdotte per la prima volta tre leggi fondamentali sull'evoluzione del software, che sono state poi estense ad otto a partire dall'anno 2011.

Con 'Evoluzione del software' si intende la necessità di manutenzione, aggiornamento e miglioramento continui per le applicazioni commerciali, affinché esse rimangano prodotti praticabili nel tempo. L'evoluzione del software è guidata dalle esigenze di business e dei consumatori, che cambiano con l'avanzare del tempo e delle possibilità tecnologiche. In particolare con un crescente utilizzo del software, diventano evidenti e necessari nuovi cambiamenti o funzionalità che devono essere introdotti nelle versioni successive degli applicativi.

A Meir Lehman è stato attribuito il merito di aver diffuso le cosiddette 'Lehman's Laws', leggi di Lehman, le quali si basano sul cambiamento continuo del software e permettono di definire il processo della sua evoluzione. In particolare viene affermato che un applicativo che non si evolve diventerà meno utile.

Le tecnologie di tipo cloud, grazie alla grande capacità di ridimensionamento dinamico delle risorse informatiche su richiesta, forniscono nuove opportunità per lo sviluppo di applicazioni scalabili e facilmente adattabili.

Grazie a ciò, gli sviluppatori sono in grado di creare nuove soluzioni a bassa latenza che permettono di gestire milioni di richieste al secondo, adattano l'utilizzo delle risorse alla loro corrente necessità e permettono una grande scalabilità.

Dal momento che questo nuovo tipo di software è stato introdotto abbastanza recentemente, non esistono, al momento, framework o pattern architetturali universalmente riconosciuti come standard nello sviluppo di sistemi distribuiti scalabili.

L'obiettivo principale di questo elaborato consiste nell'implementazione di un'applicazione orientata agli eventi, che si basa sui principi di *Event Sourcing, Command Query Responsibility Segregation e Domain Driven Design*, descritti in seguito.

Specificatamente, questo lavoro di tesi è iniziato dallo studio dello stato attuale dei sistemi distribuiti, in particolare focalizzandosi su modelli basati sugli eventi, per continuare nell'indagine dei principi introdotti da Greg Young [1] e Martin Fowler [2] e scaturire nella ricerca ed implementazione di un applicativo che possa implementare queste tecniche.

Le aspettative di questo studio sono che questi nuovi metodi di implementazione di applicativi che si basano sugli eventi possano portare, in primo luogo, a miglioramenti nelle prestazioni, ed, inoltre, ad una maggior facilità di implementazione. Infatti, con il rapido incremento nell'utilizzo di architetture a microservizi è sempre più comune incontrare architetture che manchino di una vera e propria struttura, ma risultino disorganizzate e di difficile manutenzione. Invece, ciò che si intende ottenere è un applicativo orientato agli eventi, che permette di mantenere quanto più disaccoppiati tra loro i vari servizi, che sia scalabile, responsivo e resiliente per quanto possibile, in modo che i tempi di risposta siano ridotti o contenuti in caso di guasti, estensibile, mantenibile, per uno sviluppo ed una manutenzione quanto più possibile semplice, osservabile e di facile utilizzo per diverse tipologie di utenti.

1.1 Obiettivo della tesi

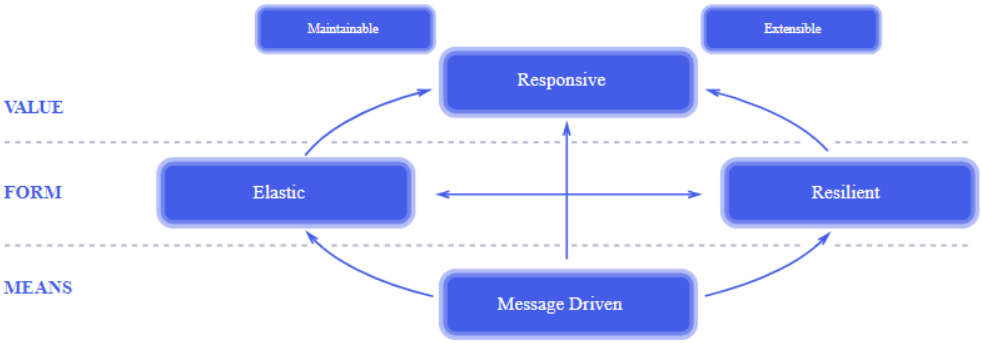


Figura 1.1: Punti principali del Reactive Manifesto.
Fonte: https://www.reactivemanifesto.org/

L'obiettivo principale nella creazione di un sistema distribuito scalabile si basa sul 'Reactive Manifesto', un documento redatto da Jonas Boner, supportato da un gruppo di sviluppatori esperti, e pubblicato il 16 settembre 2014 [3]. In questo documento vengono introdotti i punti principali che dovrebbero essere presenti all'interno di un'architettura reattiva.

In particolare, all'interno del manifesto, viene spiegato come *"Enti ed aziende operanti in vari settori stanno sperimentando in modo indipendente dei pattern architetturali similari per forgiare sistemi software. Questi sistemi si dimostrano robusti, resilienti e flessibili, e sono in grado di soddisfare al meglio le esigenze delle applicazioni software moderne."* [3]. Questi cambiamenti avvengono poiché, rispetto ad alcuni anni fa, in cui una grande applicazione richiedeva decine di server, lunghi tempi di risposta e grandi difficoltà per la manutenzione, negli ultimi tempi queste stesse applicazioni vengono distribuite su cluster basati su cloud, che eseguono migliaia di processori multi-core.

Dal momento che la tecnologia si sta evolvendo, le richieste di oggi non possono più essere soddisfatte tramite l'utilizzo di architetture software precedenti.

In particolare il Reactive Manifesto (figura 1.1) introduce la necessità di avere sistemi che siano:

- *Responsivi*: i sistemi devono permettere, per quanto possibile, risposte tempestive, riducendo al minimo i tempi di risposta.
- *Resilienti*: è necessario che il sistema risponda anche in caso di guasti o errori, un guasto su una singola componente di un sistema non deve comprometterne la sua totalità.
- *Flessibili*: il sistema deve essere in grado di adattarsi alla frequenza di richieste, incrementando o decrementando al bisogno le risorse.
- *Orientati ai messaggi*: la comunicazione tra gli elementi dell'architettura deve avvenire tramite scambio asincrono di messaggi. Questi messaggi vengono scambiati tramite il meccanismo delle code, e perciò tramite l'utilizzo di message broker.

Questo tipo di sistemi viene chiamato "reattivo", e permettono una maggior flessibilità e scalabilità, in aggiunta a permettere una maggior tollerabilità al fallimento.

L'obiettivo principale di questa tesi è la ricerca di applicativi che siano quanto più Responsivi, Resilienti, Flessibili ed Orientati ai messaggi, così come vengono descritti i sistemi reattivi.

Inoltre, in aggiunta ai principi esposti nel Reactive Manifesto, la necessità è quella di avere sistemi che siano quanto più:

- *Estensibili e mantenibili*: si richiede di poter introdurre miglioramenti il più rapidamente possibile, oltre chè di poter sviluppare nuove funzionalità e possibili correzioni di bug con il minor sforzo possibile.
- *Osservabili*: si richiede la possibilità di sapere ciò che sta succedendo in un preciso momento o ciò che è successo in qualsiasi momento precedente. Questa richiesta è dettata dalla richiesta di manutenibilità e analisi di un sistema.
- *Facili da trattare*, per tutti i tipi di utenti.

Quando si vuole sviluppare un sistema di tipo reattivo, è necessario prendere in considerazioni tutti questi aspetti, che sono fortemente collegati tra loro, come si può vedere dal grafico in figura 1.2.

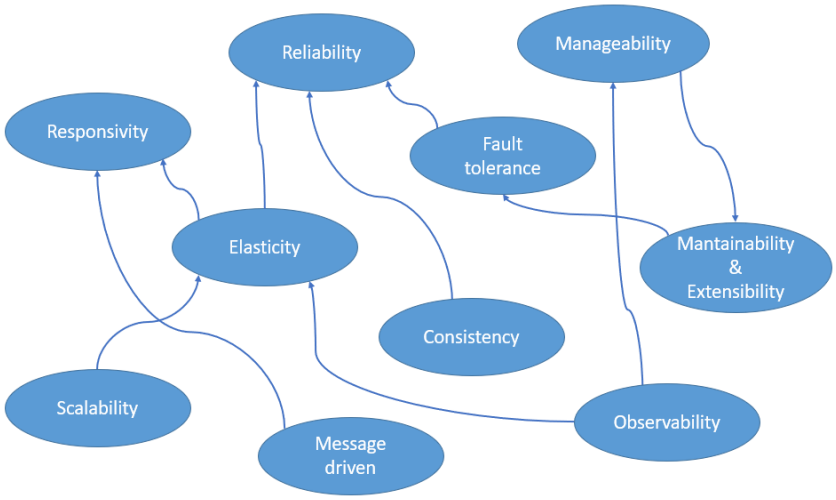


Figura 1.2: Punti principali per un sistema di tipo reattivo

Capitolo 2

Architetture: dall'eredità agli approcci moderni

Per molti anni l'architettura monolitica è stata un punto di riferimento molto importante per lo sviluppo software. In seguito, con l'emergere di nuove tecnologie, è stata sovrastata dalla diffusione del modello a microservizi e delle cosiddette applicazioni moderne.

2.1 Architetture monolitiche

Un'architettura monolitica è il modello unificato tradizionale per lo sviluppo software.

Il termine 'monolitico', ispirato alla figura del monolite, in questo contesto significa 'composto di un unico pezzo', che allude al fatto che un software monolitico sia progettato come una singola unità in esecuzione, come un singolo eseguibile, per il quale i suoi componenti non possono essere separati tra loro, né tantomeno riutilizzati senza un aggiornamento integrale dell'applicazione.

Di conseguenza, un software monolitico è progettato per essere autonomo, dove i componenti sono strettamente accoppiati ed ognuno di essi deve essere presente per l'esecuzione o la compilazione del codice e per l'esecuzione del software.

Dal momento che le applicazioni monolitiche contengono più componenti legati tra loro, uno dei principali vantaggi è la possibilità di svolgere in maniera più rapida ed efficiente fasi di test e debug del software, poiché si trova tutto sotto il controllo dello sviluppatore. Inoltre, un'architettura di tipo monolitico offre la possibilità di comunicare con le diverse parti di un'applicazione con chiamate a metodi, tramite memoria condivisa o passaggio di messaggi. Tutti questi metodi permettono una latenza molto inferiore rispetto alle chiamate di tipo HTTP o RPC. Infine, l'architettura monolitica rappresenta la metodologia di design più semplice

ed efficiente per la creazione di un'applicazione capace di eseguire simultaneamente più funzioni, poiché il suo design non prevede alcuna complessità di comunicazione tra i componenti.

Poiché un design di tipo monolitico mostra evidenti limiti quando si presenta la necessità di sviluppare e mantenere applicazioni moderne, è necessario menzionare alcuni svantaggi portati da questa architettura.

Un primo svantaggio riguarda le difficoltà che si possono incontrare durante l'aggiornamento di un software monolitico. In particolare, l'aggiornamento del software comporta il totale aggiornamento dell'applicazione, e non si presta alle logiche della distribuzione continua (CI/CD). Inoltre, è necessario menzionare una mancanza in termini di agilità, poiché questo tipo di architettura prevede l'utilizzo di un singolo linguaggio per il suo sviluppo, ed è necessario che tutti gli sviluppatori conoscano il funzionamento complessivo della totale applicazione. Questo ultimo aspetto può portare ad evidenti criticità qualora fosse necessario aggiornare un software monolitico, poiché si necessiterebbe di competenze e linguaggi di programmazione specifici.

2.2 Architetture a microservizi

L'architettura a microservizi nasce con lo scopo di superare i limiti imposti dall'architettura di tipo monolitico di fronte all'esigenza di nuove metodologie di sviluppo, come DevOps e Agile.

Un'architettura di tipo a microservizi, differentemente con quanto visto in precedenza per l'architettura di tipo monolitico, si basa sulla decomposizione del monolite, dovuta al disaccoppiamento dei suoi componenti. Infatti, applicazioni di grandi dimensioni possono essere separate in componenti più piccole, ognuna gestita in maniera indipendente ed ognuna con le proprie responsabilità. Non è più necessario che venga utilizzato lo stesso linguaggio di programmazione per tutti i componenti dell'architettura, poiché non è più essenziale una dipendenza tecnologica tra le varie funzioni che permettono la creazione dell'applicazione. Questo permette di poter scegliere diverse tecnologie, ad esempio diversi linguaggi di programmazione o diversi tipi di database, utilizzando ciò che si reputa più adatto alle funzionalità richieste ad ogni servizio. Questo consente di rendere i vari componenti software riutilizzabili ed interoperabili grazie all'utilizzo di interfacce di servizio.

Ogni servizio all'interno di una 'Service-Oriented Architecture' incarna il codice e i dati necessari per diventare una funzionalità completa e discreta, mentre *"le interfacce di servizio forniscono un accoppiamento libero, il che significa che possono essere chiamate con poca o nessuna conoscenza di come viene implementato il servizio sottostante, riducendo le dipendenze tra le applicazioni"* [4].

All'interno di un'architettura a microservizi, inoltre, non è necessario che venga aggiornata l'intera applicazione ogniqualvolta si desideri applicare una modifica.

Per il supporto di questo tipo di architettura, diversi provider mettono a disposizione delle tecnologie specifiche, come ad esempio i cosiddetti 'container', che permettono di eseguire varie applicazioni in maniera totalmente indipendente tra loro. E' necessario precisare che non è una prerogativa delle applicazioni di tipo a microservizi l'utilizzo di queste tecnologie, ma esse potrebbero anche essere utilizzate con applicazioni monolitiche, collocando l'intera applicazione in un unico container. E' evidente, però, che questo tipo di tecnologia è stato concepito per situazioni più complesse rispetto ai monoliti.

Infine, dal momento che i servizi sono indipendenti tra loro, questo approccio permette un migliore isolamento degli errori, per cui, in caso di un errore in un servizio, l'intera applicazione non smetterà di funzionare correttamente.

I servizi vengono esposti utilizzando protocolli di rete standard, come SOAP (Simple Object Access Protocol)/HTTP o Restful HTTP (JSON/HTTP), per inviare richieste di lettura o modifica dei dati. La governance dei servizi controlla il ciclo di vita per lo sviluppo e nella fase appropriata i servizi vengono pubblicati in un registro che consente agli sviluppatori di trovarli rapidamente e riutilizzarli per assemblare nuove applicazioni o processi aziendali.

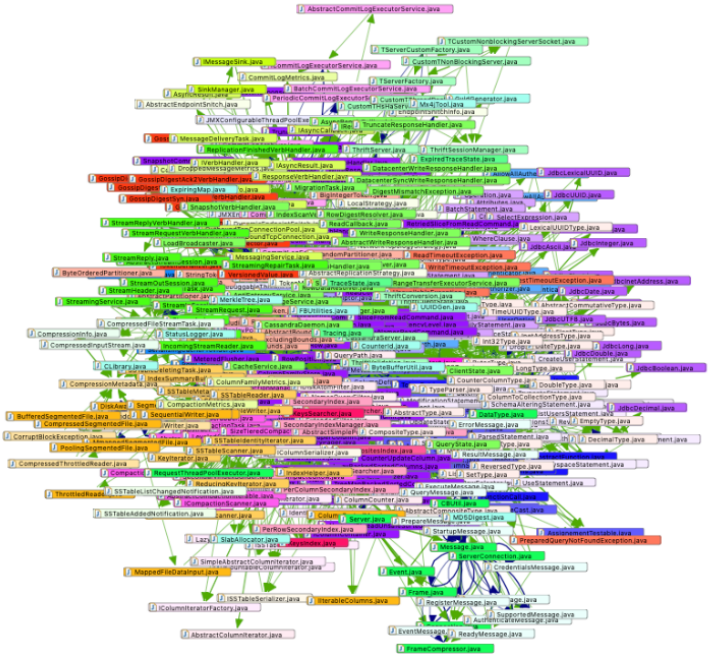


Figura 2.1: Rappresentazione di una Big Ball of Mud.
Fonte https://blog.hello2morrow.com/2021/05/analyzing-software-with-advanced-visualizations/#more-1242

Con il rapido aumento dell'utilizzo dei microservizi, è nato il concetto di 'Big Ball of Mud'. Una 'Big ball of mud', rappresentata in figura 2.1, è descritta come *"una giungla di codici di spaghetti, strutturata a casaccio, tentacolare, sciatta, con nastro adesivo e filo per balle. Questi sistemi mostrano segni inequivocabili di crescita non regolamentata e ripetute ed espedienti di riparazione. Le informazioni vengono condivise promiscuamente tra elementi distanti del sistema, spesso al punto che quasi tutte le informazioni importanti diventano globali o duplicate"* [5].

Ciò che accade è il fatto che il software manchi di una vera e propria struttura, l'architettura non è distinguibile ed è per questo che il codice appare disorganizzato e in certo casi privo di senso. Questo fenomeno può accadere, ad esempio, nel caso in cui vengano aggiunte nuove funzionalità senza pensare a come queste dovrebbero interagire in modo modulare e gestibile tra loro.

2.3 Architetture orientate agli eventi

Quando vengono sviluppati microservizi e sistemi di tipo distribuito, è necessario che i componenti siano coordinati e possano comunicare tra loro. Proprio per questo motivo si può introdurre un architettura orientata agli eventi.

Questo stile architetturale si basa sugli eventi per l'attivazione e la comunicazione tra servizi disaccoppiati, dal momento che le applicazioni possono rilevare, rispondere o pubblicare eventi. Un evento rappresenta un cambiamento di stato o un aggiornamento nel sistema.

Uno dei vantaggi di questo tipo di architettura consiste nella possibilità di disporre di un sistema disaccoppiato, il che significa che i componenti ed i servizi non hanno conoscenze approfondite l'uno dell'altro, per cui sono interoperabili tra loro e permettono una grande scalabilità. Inoltre, la totale applicazione non fallirà nel caso di un errore in uno di questi microservizi. L'utilizzo di questa architettura basata sugli eventi è molto comune tra le applicazioni moderne create con microservizi.

Il concetto di 'architettura basata sugli eventi' verrà approfondito in seguito, ma, per comprendere meglio cosa si intende con 'Architettura orientata agli eventi' si può introdurre l'esempio di una piattaforma di vendite. Nel momento in cui viene effettuato un ordine da parte di un cliente, è necessario che quest'ultimo riceva un'email di conferma dell'ordine, o un messaggio di testo (come un SMS) o similari. In una architettura orientata agli eventi ognuna di queste azioni è disaccoppiata dall'azione stessa di effettuare l'ordine, per cui esisteranno componenti differenti ed indipendenti che gestiranno azioni differenti, non ci saranno interferenze tra i componenti ed i loro ruoli.

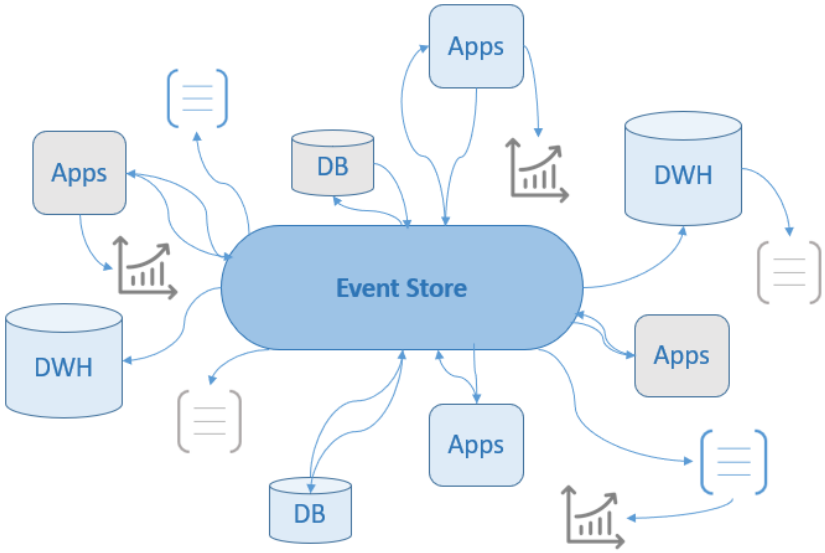


Figura 2.2: Rappresentazione di un'architettura orientata agli eventi

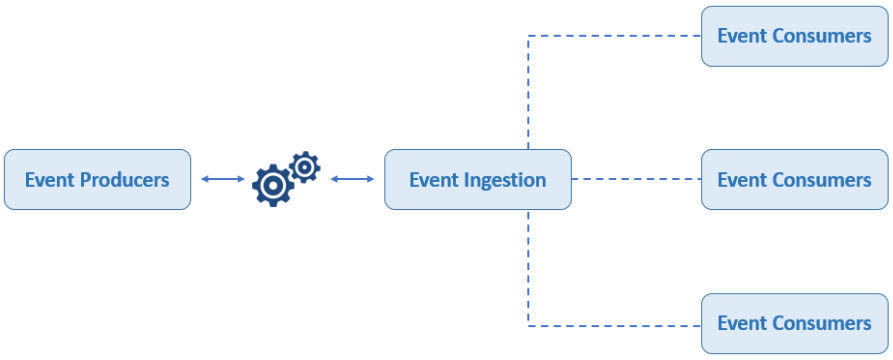


Figura 2.3: Rappresentazione di un'architettura orientata agli eventi

Capitolo 3

Event Sourcing e Command Query Responsibility Segregation (CQRS)

3.1 Introduzione all'Event Sourcing

Il termine *'Event Sourcing'* è stato coniato da Martin Fowler nel 2005 [2], e descrive un metodo alternativo per mantenere la persistenza dei dati orientata agli eventi. In particolare, l'idea principale consiste nell'avere un modo alternativo per mantenere tutti i cambiamenti avvenuti all'interno di un'applicazione sotto forma di eventi, in contrasto con i metodi popolari di mantenimenti dello stato corrente dell'applicazione.

Infatti, mentre i software maggiormente utilizzati al momento memorizzano soltanto lo stato attuale della nostra applicazione, mantenendolo persistente grazie a sistemi di database, nel caso dell'Event Sourcing, ogni evento, cioè un cambiamento avvenuto nel nostro sistema, è mantenuto all'interno di un log ordinato e autonomo di eventi, anche chiamato Event Store.

Con l'approccio tradizionale i dati vengono letti o modificati da 4 operazioni, Create (Creazione), Read (Lettura), Update (Aggiornamento), Delete (Cancellazione), generalmente definite tramite 'operazioni CRUD', invece, grazie al concetto di Event Sourcing, ogni cambiamento nello stato dell'applicazione viene definito come un evento e viene mantenuto persistente, in modo che si possa, in qualunque momento, ricreare lo stato corrente dell'applicazione.

Terminologia

- *Domain Event (Evento)*

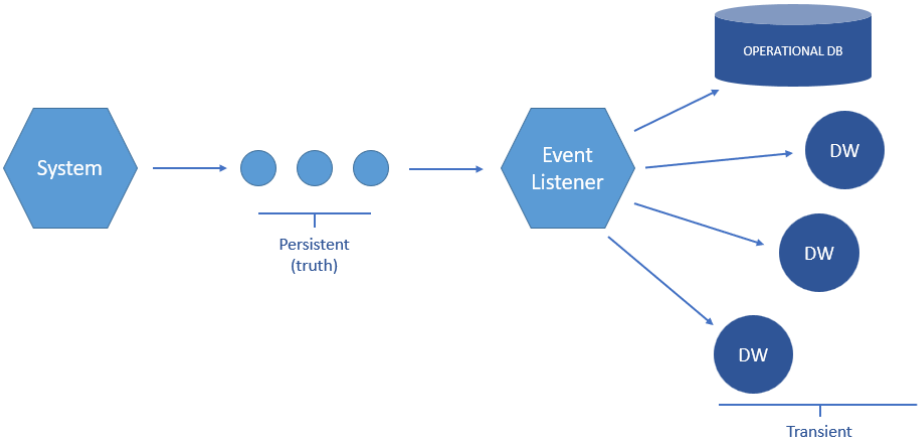


Figura 3.1: Event Sourcing in un'unica figura.

L'evento è una parte fondamentale dell'Event Sourcing, ed è relativo ad una transizione di stato della nostra applicazione. In particolare, un evento descrive un cambiamento avvenuto nel sistema, che è risultato in una modifica dello stato del sistema. É necessario menzionare alcune caratteristiche fondamentali di un evento:

a. Ogni evento è un'istanza al tempo passato, perciò il cambiamento dello stato del sistema è già stato implementato.

b. Ogni evento è immutabile, dal momento che descrive qualcosa che è già accaduto nel sistema, perciò non può essere modificato né cancellato. L'unica possibilità per annullare una modifica è pubblicare un evento successivo che la compensi.

c. Ogni evento ha una singola sorgente (publisher) da cui viene pubblicato, ma possono esistere più ricevitori (subscribers) che lo processano.

d. Il nome di un evento deve descrivere l'intento della modifica, e deve essere definito al passato.

- *Event Store*

  L'Event Store è lo spazio di archiviazione utilizzato per salvare gli eventi, al cui interno troviamo l'elenco di tutti gli eventi nell'ordine in cui si sono effettivamente verificati. È importante tenere in considerazione il fatto che questo database sia definito in modalità append-only, e per questo motivo i dati al suo interno non possono essere né modificati né cancellati. All'interno di un Event Store, per ogni evento sono necessari:

  a. Token: definisce il numero identificativo dell'evento

b. Stream Id: identifica a quale Event Stream l'evento appartiene. Solitamente, all'interno di un sistema, esistono più Event Stream, ognuno dei quali potrebbe corrispondere ad un'entità o aggregato. Questa colonna può anche essere definita come aggregate id.

c. Payload data: identifica i dati interni ad ogni evento.

d. Timestamp: identifica la data di ricezione dell'evento.

Un Event Store affidabile è un database che non subisce modifiche esplicite. Con modifica esplicita si intende una modifica diretta sulla struttura o sull'informazione degli eventi o degli Event Stream.

- *Event Stream*

  L'Event Stream viene definito come l'insieme di tutti gli eventi che sono associati ad un aggregato specifico. Può essere visto come una lista di eventi, nuovamente in modalità append-only.

- *Comando*

  Quando si parla di comandi, si intendono le richieste di modifica dei dati. Ogni comando viene processato da un Command Handler, che è incaricato di validare il comando. Nel caso in cui il comando sia validato, il che significa che è stato definito correttamente ed è consistente con lo stato corrente dell'aggregato a cui fa riferimento, viene creato un nuovo evento e aggiunto all'Event Store. Invece, nel caso in cui il comando non passi la validazione, a causa di un'incorretta definizione, oppure poiché applicandolo non verrebbe mantenuta la consistenza rispetto ai comandi precedentemente ricevuti, viene lanciata un'eccezione.

- *Aggregato*

  Il termine aggregato proviene dal Domain Driven Design e descrive *"un gruppo di oggetti associati, che possiamo trattare come un'unità per lo scopo di modifica dei dati"* [6]. Ogni aggregato è definito da un 'aggregate root', cioè un identificativo specifico, e rappresenta lo stato corrente di un oggetto. In particolare, nel dominio dell'Event Sourcing, un aggregato viene utilizzato per definire l'entità a cui viene associato un evento.

  Ogni aggregato è sempre mantenuto coerente rispetto a tutti gli eventi precedentemente pubblicati.

  Una caratteristica importante di questo tipo di oggetto è l'atto di ricostruzione dello stato corrente in cui si trova ogni qual volta in cui riceva un nuovo comando. Infatti, per ogni richiesta di modifica dei dati, vengono letti tutti gli eventi precedentemente pubblicati per ricreare il contesto in cui si trova l'oggetto in quell'istante.

- *Proiezione*

  Le proiezioni sono un modello fondamentale nell'implementazione di un sistema basato sull'Event Sourcing, all'interno del quale troviamo tutte le modifiche persistenti che si verificano nella nostra applicazione, definite come una sequenza di eventi. Dal momento che lo 'stream' di eventi è necessario per ricostruire lo stato consistente di ogni entità, in modo da poter verificare e gestire eventuali nuove richieste che potrebbero arrivare al sistema, e, con l'aumentare del numero dei dati, potrebbe risultare molto voluminoso, vengono utilizzate le proiezioni per memorizzare tutti quei valori che potrebbero servire in eventuali query richieste al sistema.

  Le proiezioni possono essere paragonate alle viste materializzate, in cui vengono definiti e memorizzati gli attributi che sono utili al fine di rispondere il più velocemente possibile ad una query richiesta al sistema.

  Inoltre, dal momento che ogni proiezione rimane in ascolto di ogni evento pubblicato riguardante la particolare entità a cui fa riferimento, i suoi attributi sono mantenuti aggiornati. D'altra parte, poiché ogni evento è asincrono, è possibile che la proiezione sia indietro rispetto all'attuale situazione del sistema, a causa della consistenza eventuale. Ogni proiezione si avvale di una coda, in cui vengono disposti gli eventi che devono essere processati, finché la proiezione stessa ha il tempo di elaborarli.

- *Snapshot*

  Poiché ogni qual volta si voglia ricostruire lo stato consistente di un aggregato, occorre recuperare dall'interno dell'Event Store tutti gli eventi accaduti e riprodurli, questo potrebbe portare ad un ritardo consistente, nel momento in cui siano presenti una grande quantità di eventi.

  Uno snapshot è una rappresentazione dello stato di un aggregato in un istante specifico nel tempo, che permette di velocizzare il processo di ricostruzione dello stato di un aggregato poiché non è più necessario leggere tutti gli eventi, partendo dall'istante iniziale, ma è sufficiente utilizzare lo snapshot congiuntamente con gli eventi che sono stati pubblicati successivamente alla creazione dello stesso.

3.1.1 Vantaggi

I vantaggi principali che scaturiscono dall'utilizzo di un design come l'Event Sourcing sono molteplici. Innanzitutto è necessario menzionare la possibilità di ricostruire, in qualunque momento, lo stato consistente del sistema a partire dalla sequenza di eventi pubblicati, che vengono salvati all'interno dell'event store. Questo permette

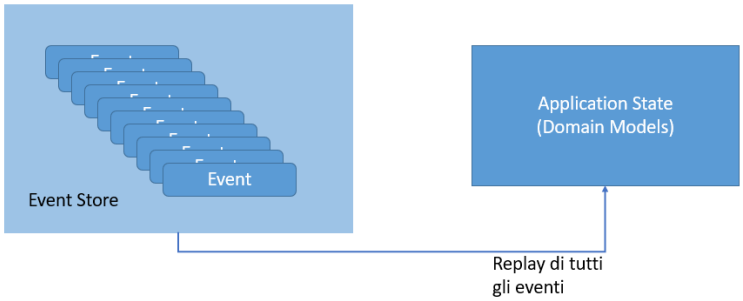


Figura 3.2: Rappresentazione del concetto di Event Sourcing

di facilitare la fase di testing e debug del sistema, poiché è sempre possibile ritornare ad un punto definito nel tempo del sistema, ricreando tutto ciò che è accaduto fino a quell'istante.

In secondo luogo, sempre legato alla facilità con cui si può ricostruire lo stato del sistema, un altro vantaggio è la semplicità con cui si possono aggiungere o modificare proiezioni per poter soddisfare ogni nostra esigenza. Infatti, è sufficiente creare una nuova proiezione che elabori tutti gli eventi storici di suo interesse. In particolare, per comprendere meglio i vantaggi esposti, si può seguire l'esempio di un database relazionale. All'interno di quest'ultimi si trovano dei registri di log di eventi immutabili, che permettono il mantenimento di tutte le modifiche applicate al database, in modo tale che si possano gestire al meglio tutte le transazioni, ripristinando lo stato precedente o replicando modifiche in caso di anomalie o azioni non volute. In sostanza, piuttosto che mantenere soltanto lo stato corrente di un sistema, memorizzando soltanto i risultati delle modifiche applicate, vengono salvate le stesse modifiche.

È importante menzionare i vantaggi portati dall'Event Sourcing anche nel caso in cui avvenga un guasto nel nostro sistema, a seguito di modifiche non corrette apportate sullo stesso. Nel caso in cui non si avesse un backup recente dei dati utilizzati, non ci sarebbe modo di ripristinarli in un sistema in cui non viene utilizzato l'Event Sourcing, ma, nel caso in cui si utilizzasse questo tipo di design, avremmo tutti gli eventi accaduti sul sistema, cioè tutte le modifiche apportate, per cui basterebbe ricreare i dati finali riproducendo tutti gli eventi.

Un ulteriore progresso nell'utilizzo dell'Event Sourcing all'interno delle applicazioni è quello di mantenere diverse versioni di uno stesso sistema. Questa possibilità è definita come 'versioning' e consiste nell'avere due versioni di uno stesso sistema l'una accanto all'altra, di cui una si occuperà di generare gli eventi, ma entrambe potranno sfruttarli ed elaborarli in totale indipendenza l'una dall'altra.

Pertanto, l'utilizzo dell'Event Sourcing e CQRS alla base della struttura di un'applicazione può portare innumerevoli vantaggi, per cui si possono ottenere

sistemi più semplici da modellare, più veloci e che portano una maggior flessibilità. È necessario, però, tenere in considerazione gli svantaggi che porta con se' questo tipo di design.

3.1.2 Svantaggi

Un primo svantaggio, di cui è necessario far menzione, è la gestione di entità con una lunga e complessa durata. In particolare, possono esistere alcune entità caratterizzate da frequenti cambiamenti di stato, per cui è necessario mantenere una grande quantità di eventi, e pertanto rielaborarli tutti ogni qual volta si debba ricostruire lo stato dell'aggregato corrispondente a questa entità. Il problema può essere in parte superato con l'utilizzo degli snapshot, che, come già riportato in precedenza, permettono di mantenere una 'fotografia' dello stato di un evento in un particolare istante di tempo. Indubbiamente è necessario definire come e quando dovrebbero essere creati questi snapshot, basandosi sul momento in cui si avrà un carico di query che potrebbero portare a rallentamenti del sistema.

In secondo luogo, è necessario affrontare il problema della consistenza eventuale. In particolare, nel momento in cui un utente richiede una modifica al nostro sistema, per cui viene inviato un comando, questo dopo essere stato verificato, verrà accettato e perciò sarà pubblicato l'evento rispettivo.

Però, dal momento che i nostri modelli di lettura si avvalgono della consistenza eventuale, un altro utente potrebbe non vedere subito la modifica e inviare, poco dopo il primo comando, un nuovo comando che agisce sullo stesso aggregato. Il comando verrà rifiutato dall'Event Store, se non può essere applicato sui dati aggiornati, e perciò sarà necessario gestire l'eccezione che verrà lanciata. È anche importante menzionare le difficoltà che si possono incontrare nel momento in cui si vogliano apportare cambiamenti alla struttura dei dati ricevuti dall'applicazione. In particolare, nonostante gli eventi siano immutabili, è possibile che si abbia la necessità di cambiare lo 'schema' degli eventi. Pertanto, si può pensare di aggiornare tutti gli eventi esistenti, oppure mantenere schemi differenti per gli stessi eventi.

Per concludere, bisogna tenere in considerazione la difficoltà che si può riscontrare nel momento in cui il numero di comandi, e pertanto anche di eventi, che vengono salvati dal nostro sistema aumenta. Nel caso in cui si debbano mantenere pochi eventi, la gestione di questi ultimi può essere semplice, ma quando si hanno numeri consistenti, la logica di elaborazione diventa onerosa molto rapidamente.

In particolare, come già menzionato in precedenza, per validare ogni comando viene ricostruito lo stato consistente dell'aggregato leggendo tutti gli eventi accaduti dall'Event Store, e ciò può diventare un 'collo di bottiglia' nel caso di numerosi eventi. Pertanto, in generale, è consigliato utilizzare sistemi basati su Event Sourcing e CQRS nel caso in cui siano richieste al sistema un numero molto maggiore di query, rispetto al numero di comandi.

3.2 Introduzione al CQRS

Il modello CQRS, che significa *'Command Query Responsibility Segregation'*, ha origine dal modello chiamato 'Command Query Segregation', proposto da Bertrand Meyer nel libro 'Object Oriented Software Construction' da lui scritto [7]. Meyer sostiene che ogni metodo all'interno di una classe dovrebbe essere definito o come una query, e pertanto restituire dati senza modificarli, o come un comando, modificando i dati ma non restituendo nulla.

L'architettura del CQRS, introdotta da Greg Young e sviluppata da Martin Fowler [1] [8], estende l'idea del CQS, applicandola a livello del sistema. In particolare, viene proposta una totale separazione delle responsabilità, dividendo la gestione delle modifiche ai dati(comandi) rispetto alla gestione delle richieste di lettura dei dati (interrogazioni o 'queries'), ottenendo così due modelli completamente separati, uno per l'aggiornamento e uno per la visualizzazione.

Ciò che si intende per 'modelli separati' è l'utilizzo di modelli a oggetti differenti, che possono sia essere eseguiti in processi logici differenti, sia su hardware separati. Possiamo introdurre molteplici variazioni in riferimento a questo nuovo modo di sviluppare un'applicazione: in primo luogo i due modelli potrebbero sia condividere lo stesso database, sia utilizzare database separati. Nel primo caso il database sarà il mezzo di comunicazione tra i modelli, nel secondo caso sarà necessaria l'implementazione di un meccanismo di comunicazione tra i due modelli.

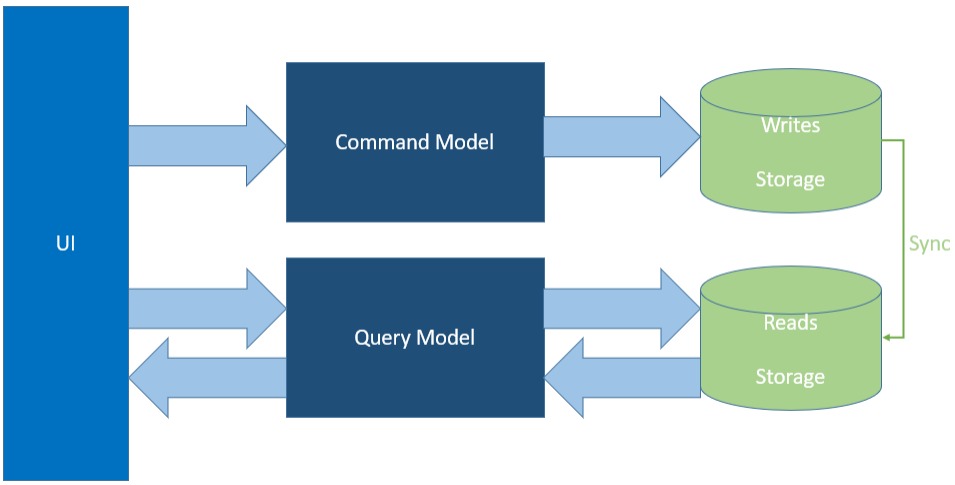


Figura 3.3: Command Query Responsibility Segregation in un'unica figura.

Lato dei comandi

Quando viene richiesto un comando, questo viene ricevuto da una specifica entità, e verrà trasformato in un evento. È necessario un gestore di comandi, che permette di memorizzare i comandi correttamente.

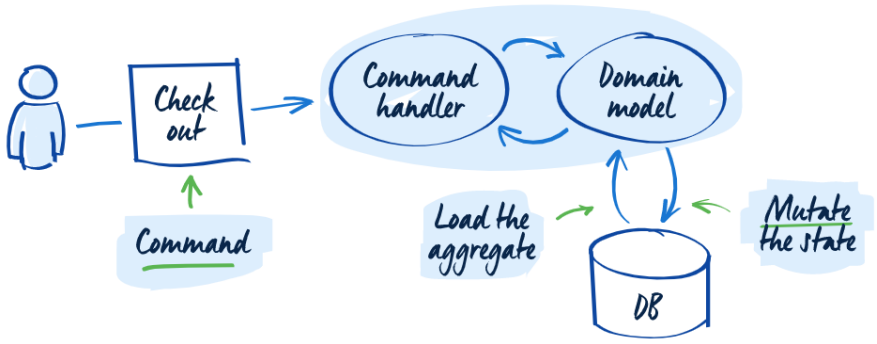


Figura 3.4: Esempio per lato di comandi nell'Event Sourcing e CQRS
Fonte: https://www.eventstore.com/blog/event-sourcing-and-cqrs

Lato delle interrogazioni o 'query'

All'interno del lato di query avremmo tutti i metodi necessari per la lettura dei dati. Ogni entità (detta proiezione) riceverà gli eventi di suo interesse in modo che possa essere mantenuta aggiornata. Come per il lato dei comandi, anche qui ci sarà un gestore di queries, che restituirà i dati all'utente.

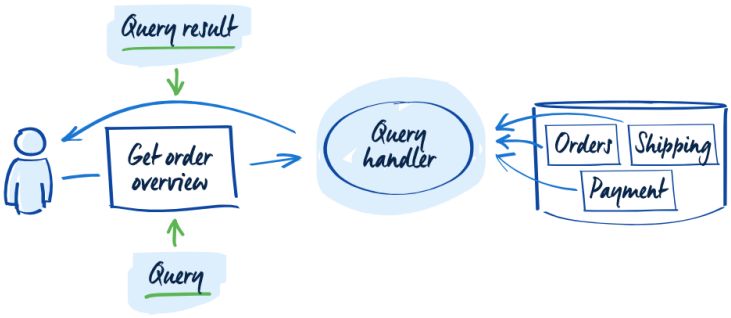


Figura 3.5: Esempio per lato delle query nell'Event Sourcing e CQRS
Fonte: https://www.eventstore.com/blog/event-sourcing-and-cqrs

3.2.1 Vantaggi

Come ogni pattern, il CQRS può essere molto vantaggioso in alcuni casi, meno in altri. In particolare, un primo vantaggio da menzionare quando si parla di CQRS

è la riduzione che si può ottenere nel carico di lavoro del modello, dovuto alla separazione del modello di lettura e scrittura, soprattutto nel caso in cui si abbia una forte disparità tra letture e scritture. Inoltre, il lato di lettura potrebbe utilizzare uno schema ottimizzato per le query, mentre il lato di scrittura ne utilizzerebbe un altro ottimizzato per i comandi. Infine, si ottiene una maggior semplicità nel garantire la sicurezza all'interno del sistema, permettendo solo alle entità corrette di eseguire modifiche sui dati.

3.2.2 Svantaggi

Indubbiamente non mancano gli svantaggi portati dall'utilizzo di un pattern come il CQRS. In primo luogo, nonostante l'idea alla base di questo tipo di pattern sia relativamente semplice, la progettazione di un'applicazione può risultare complessa, soprattutto se il CQRS viene utilizzato in concomitanza con altri pattern, come l'Event Sourcing. La separazione dei database di lettura e scrittura porta ad ottenere una coerenza soltanto eventuale, per la quale i dati richiesti da una lettura potrebbero risultare obsoleti. Inoltre, l'utilizzo di più database comporta un aumento dei costi necessari per la creazione ed il mantenimento del sistema.

3.2.3 Conclusioni sul CQRS

In conclusione, il CQRS è un pattern architetturale che fornisce stabilità e scalabilità alle applicazioni, migliorando le prestazioni complessive. Si possono considerare alcuni scenari in cui è opportuno l'utilizzo del CQRS:

- Domini collaborativi in cui sono richieste molte letture ai dati in parallelo, in cui viene richiesta un'ottimizzazione delle prestazioni delle letture rispetto alle prestazioni per la parte di scrittura. In generale è consigliato in tutti quei sistemi in cui il numero di operazioni di lettura è molto maggiore rispetto al numero di operazioni di scrittura.

- Domini in cui si prevede che il sistema si possa evolvere e quindi contenere versioni differenti nel tempo.

- Domini in cui è richiesto che due team di lavoro differenti si occupino del modello di lettura e scrittura in maniera indipendente. Questo tipo di pattern non è raccomandato nel caso in cui si avesse un dominio semplice, poiché potrebbe introdurre complessità non necessaria, oppure nel caso in cui sia sufficiente una semplice interfaccia di tipo CRUD.

3.3 CQRS ed Eventual Consistency

Come già anticipato in precedenza, un'applicazione che fa uso del pattern CQRS prevede la creazione di due modelli completamente separati, uno per la lettura e uno per la scrittura dei dati. Se da una parte il modello di scrittura avrà come garanzia una totale coerenza, poiché i dati vengono aggiornati nel momento in cui un comando, dopo essere arrivato ad un'entità, viene accettato, d'altra parte, nel modello di lettura, la coerenza può essere soltanto di tipo eventuale. Infatti, questo permette di avere più di un gestore di eventi, ognuno dei quali aggiorna un modello di lettura in maniera concorrente l'uno con l'altro, in modo che ogni modello di lettura sia indipendente rispetto agli altri. Questo non permette di avere una coerenza puntuale, ma soltanto eventuale.

Ciò che è importante sottolineare quando si parla di consistenza eventuale è il fatto che i dati che vengono letti non saranno mai scorretti, ma potrebbero essere vecchi. Infatti, i dati letti potrebbero non essere ancora stati aggiornati, ma sono stati corretti in precedenza, prima che venisse richiesto l'aggiornamento. Con il concetto di consistenza si può introdurre il teorema CAP.

3.3.1 CAP Theorem

Il teorema CAP fu introdotto da Eric Brewer nel 1998 [9] e afferma che soltanto due su tre proprietà possono essere pienamente soddisfatte all'interno di un sistema distribuito:

- *Consistenza*

  un sistema consistente richiede che tutti i suoi nodi vedano gli stessi dati nello stesso momento. Per questo motivo il sistema potrà o funzionare correttamente oppure non funzionare affatto, poiché ad ogni richiesta di lettura si potrà rispondere soltanto o con dati corretti oppure con un errore.

  In particolare, i database sono molto utilizzati per garantire la consistenza. Infatti, si concentrano sulle proprietà ACID (Atomicità, Consistenza, Isolamento e Durabilità), che garantiscono la coerenza e forniscono isolamento, cioè la necessità di ogni transazione che viene eseguita sul database di essere indipendente rispetto alle altre transazioni.

- *Disponibilità*

  un sistema disponibile garantisce una risposta per ogni query richiesta, che potrà essere di successo o di fallimento rispetto all'azione richiesta.

- *Tolleranza di partizione o scalabilità*

  il sistema garantisce il suo funzionamento nonostante possibili ritardi o perdite di messaggi.

Il vantaggio che si ottiene nell'utilizzo del CQRS come pattern per un sistema è una grande disponibilità e scalabilità a discapito della consistenza, che come già anticipato non potrà essere puntuale.

3.4 Cenni sul Domain-Driven-Design

Il concetto di *Domain-Driven-Design* è stato introdotto da Eric Evans nel libro dallo stesso titolo ' Domain-Driven Design: Tackling Complexity in the Heart of Software' [10].

Secondo la definizione ufficiale, il DDD *"is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains"* [10]. Può essere definito come un approccio allo sviluppo di software, che permette di ridurre la complessità non necessaria e offre metodologie per la gestione di quest'ultima. Vengono definiti i pattern strategici, le linee guida ad alto livello che possono essere utilizzate per la progettazione di grandi sistemi e tutti gli elementi costitutivi a livello di una singola classe per la definizione della logica richiesta.

In particolare Evans descrive quattro concetti principali del Domain Driven Design:

- *Dominio*

  Il dominio di un software è l'area in cui un utente lavora con lo stesso.

- *Modello*

  E' un'astrazione di un sistema che descrive alcuni aspetti di un dominio. L'applicazione di un modello consente di risolvere alcuni problemi relativi al dominio.

- *Ubiquitous Language*

  La creazione di un linguaggio comune, costruito attorno al modello di dominio.

- *Contesto*

  L'ambiente in cui determinate affermazioni o parole assumono determinati significati. Tutte le affermazioni posso essere comprese soltanto all'interno del contesto.

Questi quattro concetti insieme formano il modello di dominio, che permette di consolidare il comportamento e i dati degli oggetti all'interno del dominio.

Al fine di creare un modello di dominio è necessario tenere in considerazione alcuni elementi fondamentali, che permettono di chiarire il modello e mantenerlo allineato con la sua effettiva attuazione.

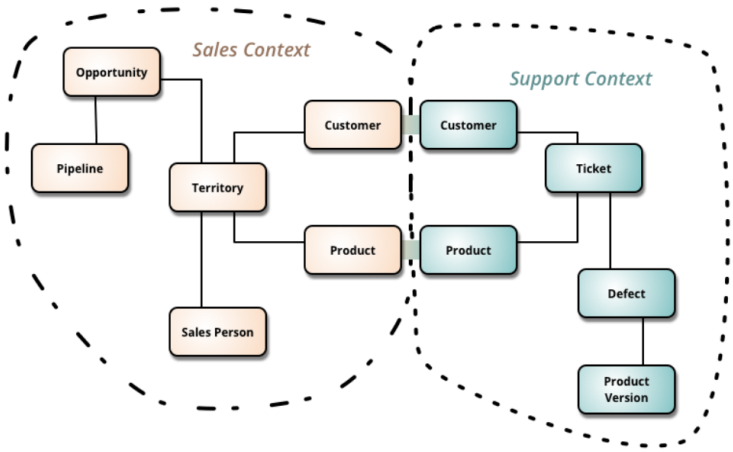


Figura 3.6: Esempio di Bounded Context per Domain Driven Design.
Fonte: https://martinfowler.com/bliki/BoundedContext.html

Gli elementi costitutivi di un modello di dominio rilevanti ai fini di questa trattazione sono: entità, aggregato, evento di dominio e oggetto di valore ('value object').

- *Entità*

  Eric Evans definisce le entità come *"not fundamentally defined by their properties, but rather by a thread of continuity and identity. An ENTITY is anything that has continuity through a life cycle and distinctions independent of attributes that are important to the application's user"* . [10] Un'entità può quindi essere definita come un'astrazione in memoria di qualcosa che cambia nel tempo, come una funzione di appartenenza temporanea, che, per un tempo t, associa un identificatore ad uno stato. Rappresenta, perciò, tutto ciò che cambia all'interno del modello di dominio.

- *Aggregato*

  Un aggregato può essere definito come un insieme di oggetti, che possono essere visti come una singola unità, con l'intento di modificare i dati. Ogni transazione all'interno del nostro sistema può coinvolgere un solo aggregato, poiché ogni aggregato accetta dei comandi specifici che arrivano dall'utente e produrrà gli eventi relativi.

- *Evento di dominio*

  Ogni 'domain event' permette di registrare qualcosa che accade all'interno del nostro sistema.

- *Value object*

  Un oggetto di valore rappresenta un insieme di attributi, ma senza un'identità concettuale. È necessario trattare gli oggetti di valori come oggetti immutabili.

Capitolo 4

Letteratura correlata

4.1 Event Store

Come descritto da Martin Fowler, un database di tipo 'schemaless' o senza schema *"allows any data, structured with individual fields and structures, to be stored in the database."* [11], cioè permette l'archiviazione di qualsiasi tipo di dato, che può essere strutturato con molteplici campi o strutture. Un database di questo genere permette di ottenere una grande flessibilità, poiché qualunque tipo di schema è accettato. Naturalmente, questo comporta che la nostra applicazione debba fare delle ipotesi sulla struttura dei dati, che verranno definiti come 'dati con schema implicito'.

In particolare, con l'utilizzo dell'Event Sourcing come pattern architetturale, è necessario un *'Event Store'*, un database che contenga tutti gli eventi passati e futuri riguardanti il nostro sistema. Questi eventi avranno uno schema di tipo implicito, che non è necessario conoscere, poiché questo tipo di database verrà utilizzato soltanto per la persistenza.

Allo scopo di scegliere il database più adatto da utilizzare come Event Store all'interno della nostra applicazione è necessario tenere in considerazione alcuni requisiti:

- *Lato di scrittura*: per quanto riguarda il modello di scrittura, è necessario che la latenza per la persistenza di un evento rimanga costante ed indipendente dalla quantità di eventi presenti. Inoltre, per una singola transazione, cioè un singolo comando eseguito, più eventi devono poter essere scritti nello stesso momento. Infine, come precedentemente detto, non deve essere definito uno schema determinato per gli eventi.

- *Lato di lettura*: la lettura di tutti gli eventi deve avvenire nello stesso ordine in cui sono stati scritti, è necessario, infatti, che il numero di sequenza degli eventi

cresca in ordine crescente, e che ogni evento abbia un numero di sequenza univoco.

Inoltre, dal momento che il numero di eventi per un singolo aggregato può crescere esponenzialmente nel breve periodo, è necessario che un database sia in grado di leggere soltanto glie eventi accaduti a partire da un preciso momento nel tempo.

Infine, nessun evento, una volta pubblicato, può essere modificato. Se viene richiesta una modifica ad un evento, si rende necessaria la pubblicazione di un nuovo evento che apporti questa modifica.

4.2 Possibili implementazioni di un Event Store

Un Event Store può essere implementato con differenti architetture, tra cui RDBMS, MongoDB, Kafka, EventStoreDB e Axon Server.

RDBMS (Relational Database Management Systems)

Il termine *RDBMS* indica un sistema di gestione per database basati sul modello relazionale. I database di tipo relazionale utilizzati come Event Store permettono di mantenere una semplicità di utilizzo, dovuto al fatto che i dati richiesti per un sistema basato su Event Sourcing sono pochi e facilmente comprensibili: basterà mantener traccia di un numero di sequenza globale, un identificatore per le diverse entità e un payload, costituito dall'evento stesso.

Un database di tipo RDBMS permette di soddisfare tutti i requisiti necessari per un Event Store, ma, tutto questo si ottiene a discapito delle performance sulla velocità effettiva di scrittura.

Il test in figura 4.1 è stato prodotto misurando il throughput di scrittura di un RDBMS mentre veniva riempito di milioni di eventi. Una volta raggiunti circa 20 milioni di eventi, le prestazioni sono dimezzate e continuano a diminuire gradualmente fino a stabilizzarsi.

MongoDB

MongoDB è uno dei database più popolari tra i database di tipo NOSQL. Come caratteristiche principali di un database di questo tipo è necessario far menzione di grande scalabilità tramite lo 'sharding' e la possibilità di avere archiviazione di tipo 'schemaless'. Tuttavia, MongoDB è un database definito 'document-based', il che significa che può supportare nativamente soltanto transazioni di documenti singoli. Questo comporta la possibilità di salvare o più eventi in un unico documento, oppure, grazie ad una nuova funzionalità, più eventi su più documenti. Mentre nel

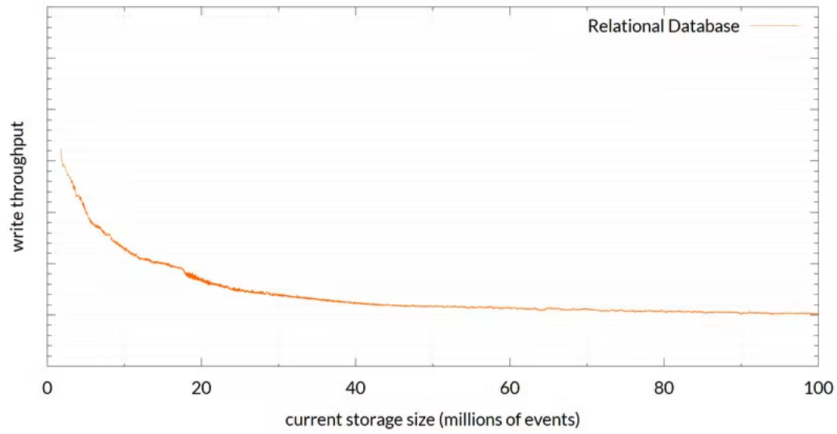


Figura 4.1: Performance di un Event Store basato su RDBMS.
Fonte https://www.axoniq.io/blog/why-would-i-need-a-specialized-event-store

caso della prima possibilità, la complessità è maggiore per la ricerca di eventi di una specifica entità, nel caso della seconda possibilità, la complessità aumenta dal momento che è necessario l'utilizzo di questa nuova funzionalità.

Kafka

Apache Kafka è una soluzione open-source di tipo publish-subscribe il cui scopo è *"la gestione di streaming di dati, anche in tempo reale, il pipelining e il replay distribuiti di feed di dati per operazioni veloci e scalabili"* [12].

Si tratta, quindi, di una soluzione in grado di elaborare e archiviare flussi di dati, grazie alla quale è possibile pubblicare o consumare messaggi su argomenti definiti. Dal momento che il concetto alla base di Apache Kafka sembra essere proprio quello di un evento, potrebbe sembrare la soluzione ideale per l'implementazione di un Event Store scalabile. In realtà, è necessario far menzione di alcuni aspetti negativi: innanzitutto, non è propriamente definito un metodo per l'archiviazione ed il recupero di eventi, considerando i requisiti sulla lettura e scrittura richiesti ad un Event Store. In particolare, nel caso si scegliesse di utilizzare un singolo argomento (topic) per la pubblicazione degli eventi, non ci sarebbero distinzioni tra eventi di differenti entità, perciò, nel caso si volessero leggere solo gli eventi relativi ad una singola entità, sarebbe comunque necessario leggere tutti gli eventi. Invece, nel caso si scegliesse di utilizzare un argomento per ogni entità, in modo da ottimizzare la lettura, sarebbe necessario ripristinare l'ordine globale degli eventi.

Introducendo la possibilità di partizionare i topic, Kafka garantisce che tutti i messaggi con la stesa chiave siano salvati nella stessa partizione, per cui basterebbe segnalare la necessità di leggere da una partizione specifica. In questo caso sarebbe

necessaria una garanzia su una distribuzione uniforme delle entità presenti nel nostro sistema.

Come ultimo svantaggio nell'utilizzo di Kafka, è necessario menzionare il fatto che applicazioni basate su Event Sourcing richiedono che un evento venga memorizzato, prima di essere pubblicato. Questo è impossibile da ottenere nel caso in cui si utilizzi Kafka come Event Store, poiché la pubblicazione e l'archiviazione diventano un unico passaggio.

EventStoreDB

Il database chiamato *'Event Store'* è una soluzione creata appositamente per applicazioni basate su Event Sourcing, perciò soddisfa tutti i requisiti che dovrebbe avere un Event Store. È scritto in .NET, ed è generalmente visto come parte dell'ecosistema .NET.

Axon Server

Axon Server combina un router di messaggi con un Event Store, in modo da fornire un'unica soluzione per l'archiviazione degli eventi e la distribuzione di messaggi tra i vari componenti di un'applicazione basata su Event Sourcing e Command Query Responsibility Segregation.

Essendo sviluppato appositamente in combinazione con Axon Framework, incontra tutti i requisiti richiesti ad un Event Store. In particolare, si può notare una grande disparità nelle performance sulla velocità effettiva di scrittura rispetto ad un RDBMS. Infatti, come si può notare in figura 4.2, il throughput rimane costante, indipendentemente dal numero di eventi archiviati nell'event store.

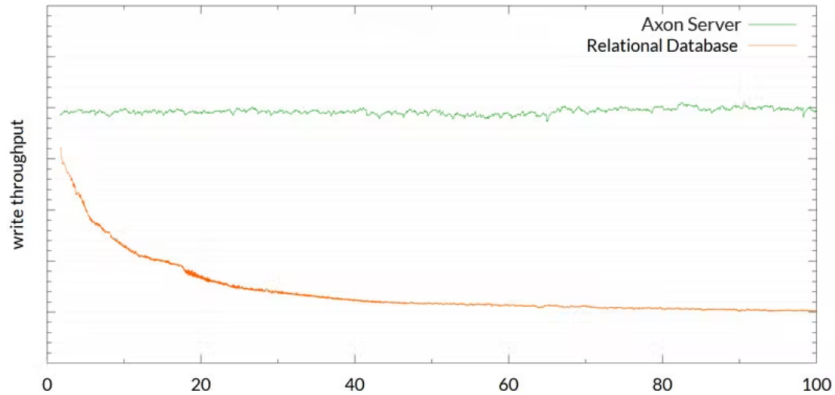


Figura 4.2: Performance di un Event Store implementato con Axon Server. Fonte https://www.axoniq.io/blog/why-would-i-need-a-specialized-event-store

4.3 JGroups

JGroups è una libreria per la comunicazione di gruppo affidabile scritta in linguaggio Java. Si basa su IP multicast, ma ne promette l'estensione grazie a:

- affidabilità: la comunicazione è resa affidabile poiché JGroups garantisce la trasmissione di messaggi senza perdite a tutti i destinatari e la ritrasmissione in casi di messaggi mancanti. Inoltre permette la frammentazione di grandi messaggi in messaggi più piccoli e il riassemblaggio a lato del destinatario. Infine garantisce l'ordinamento dei messaggi, l'ordine con cui vengono spediti all'origine sarà lo stesso di ricezione a destinazione, e l'atomicità, poiché un messaggio verrà ricevuti da tutti i destinatari o da nessuno di essi.

- appartenenza a gruppi: jGroups garantisce la conoscenza di tutti i membri di un gruppo e la gestione delle notifiche nel momento in cui un nuovo membro si unisce al gruppo, un membro precedente lascia il gruppo o un membro esistente si arresta in modo anomalo.

Pertanto JGroups permette l'estensione della trasmissione affidabile di messaggi unicast al multicast. Inoltre, fornisce uno stack di protocollo flessibile, caratteristica molto importante che permette l'adattamento a diversi requisiti di diversi tipi di applicazioni o caratteristiche di rete, poiché *"consente agli utenti di mettere insieme stack personalizzati, che vanno da stack inaffidabili ma veloci a stack altamente affidabili ma più lenti"* [13].

Quando viene utilizzato JGroups, i componenti entrano nei gruppi in modo da poter inviare dei messaggi agli altri componenti. Per poter aderire ad un gruppo, è necessaria la creazione di un canale e la connessione ad esso tramite il nome del gruppo a cui ci si vuole unire. Infatti, l'insieme di tutti i canali con lo stesso nome formeranno un gruppo.

Una volta connesso ad un gruppo, un componente può inviare e ricevere messaggi e, nel caso si incontri la necessità di scollegarsi, basterà che si disconnetta al canale.

Nel caso un componente voglia unirsi a più gruppi contemporaneamente, sarà necessario creare più canali, poiché un canale consente di collegare soltanto un client alla volta.

All'interno di un gruppo, ogni membro può recuperare un elenco di indirizzi, ognuno dei quali identifica univocamente un canale, nel caso volesse inviare un messaggio unicast ad un altro membro del gruppo oppure multicast a tutti i membri del gruppo.

Le proprietà di un canale sono tipicamente definite all'interno di un file di tipo 'XML', ma JGroups permette la configurazione dei canali anche tramite stringhe, URI, alberi DOM o in maniera programmatica.

Capitolo 5

Axon Framework

5.1 Axon Framework

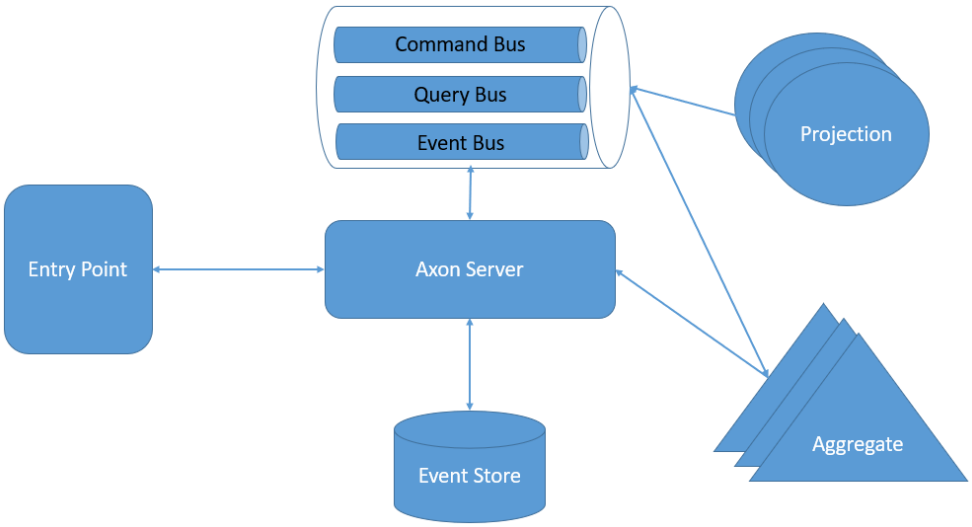


Figura 5.1: Rappresentazione di Axon Framework ed Axon Server.

Il modello Command Query Responsibility Segregation (CQRS) è fondamentalmente diverso rispetto ad una architettura a strati, ed è per questo motivo che gli sviluppatori faticano a navigare questo tipo di architettura. Per ovviare al problema è stato introdotto *Axon Framework*, un framework Java, che permette di supportare gli sviluppatori nell'applicazione del modello architetturale CQRS e dell'Event Sourcing. In particolare, essendo questo framework basato sui principi architetturali del Domain Driven Design e CQRS, fornisce gli elementi costitutivi necessari per questo tipo di applicazioni e permette la creazione di applicazioni

scalabili ed estensibili, mantenendo la coerenza delle applicazioni nei sistemi distribuiti. Inoltre, Axon fornisce supporto per le annotazioni, che possono essere usate per la creazione del modello di dominio e dei 'listener' di eventi , senza dover utilizzare la logica specifica di Axon.

Axon Framework permette la separazione dei messaggi in tre categorie distinte:

- Comando: richiesta di modifica del sistema
- Query richiesta di informazioni al sistema
- Eventi: rappresentano il verificarsi di qualcosa che potrebbe essere importante per altri componenti.

Axon Framework supporta, inoltre, la configurazione di Spring Framework e l'inserimento di dipendenze.

5.2 Vantaggi portati da Axon Framework

L'utilizzo di Axon Framework all'interno di questo progetto porta ad alcuni vantaggi. In primo luogo il framework supporta tutti i dettagli fondamentali dei concetti di Command Query Responsibility Segregation ed Event Sourcing, ed è facilmente integrabile anche nel caso di un progetto già consolidato, poiché non impone l'uso di implementazioni specifiche, ma piuttosto, è aperto a varie progettazioni e scopi di un applicazione.

Inoltre, questo framework garantisce scalabilità, poiché i componenti logici sono separati all'interno dell'architettura e sono accoppiati tra loro grazie ad un bus di messaggi asincrono. Questo permette una separazione fisica dei componenti tra server differenti, e, pertanto, di realizzare una scalabilità lineare.

In aggiunta, dal momento che le applicazioni verranno modificate più volte durante la loro vita, Axon Framework garantisce che le modifiche apportate in un componente non influiscano direttamente sul comportamento di altri componenti, grazie al 'loose coupling', cioè il promuovere legami deboli tra essi. Questo garantisce anche una facilità nell'aggiunta di nuovi elementi applicativi, poiché quelli già esistenti non devono essere modificati.

Infine, dal momento che si richiede la distinzione tra i componenti di tipo 'query' o lettura ed i componenti di tipo 'update' o scrittura, la loro struttura è ottimizzata per questi scopi differenti, perciò i componenti di query avranno accesso più rapido ai dati già strutturati per lo scopo.

E' vantaggioso menzionare la completezza e l'abbondanza di documentazione presente per un framework che permette l'implementazione del Command Query Responsibility Segregation in Java, e la facilità con cui un progetto basato su questo framework possa essere testato ed esteso a vari use-cases.

Capitolo 6

Sviluppo di un'applicativo basato su Event Sourcing e CQRS

6.1 Definizione del problema

Questo progetto si baserà sullo sviluppo di un'applicazione che possa spiegare nel dettaglio gli argomenti introdotti in precedenza, Command Query Responsibility Segregation ed Event Sourcing.

L'obiettivo dell'applicativo sviluppato è quello di realizzare una nuova versione di un applicativo correntemente utilizzato in una società attiva nel settore automotive, che porti a miglioramenti in termini di prestazioni, di scalabilità e di manutenibilità.

Il primo passo è stato quello di definire, tramite la tecnica dell'Event Storming, tutti gli elementi necessari per il corretto funzionamento dell'applicativo, a partire dalla necessità di definire come una centralina installata su un motore venga associata ad un veicolo, fino a i messaggi che vengono inviati da questa stessa centralina e che riguardano lo 'stato di salute' del veicolo stesso, da quando questo viene accesso, a quando si accendono delle spie riguardanti problemi al motore, fino al suo spegnimento.

Come è mostrato in figura 6.1, si è deciso di dividere l'applicativo in più elementi, ognuno dei quali fisicamente separato rispetto agli altri ma strettamente correlato nel ruolo.

Innanzitutto, il primo elemento che è stato implementato riguarda un 'veicolo', che consiste nella centralina installata sul motore di un veicolo, e permette l'invio di vari tipi di messaggi, descritti nella sezione 6.2. Per questo elemento sono state implementate le entità chiamate *'Dongle'* (centralina) ed *'Engine'* (motore) ed i

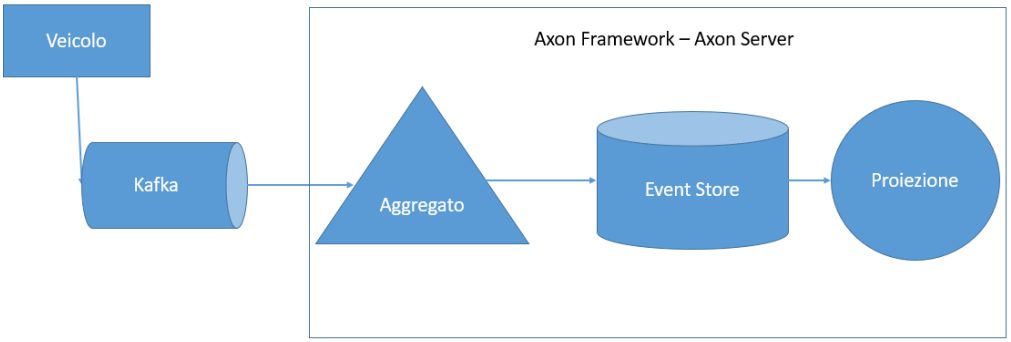


Figura 6.1: Rappresentazione dell'implementazione dell'applicativo

relativi aggregati *'DongleAggregate'* ed *'EngineAggregateEntity'*, in modo da poter rappresentare gli oggetti principali alla base di tutto il progetto.

In seguito, si è deciso di utilizzare Kafka come broker dei messaggi inviati da una centralina. In particolare, Kafka viene utilizzato per pubblicare nuovi comandi, cioè richieste di modifica dello stato di un'entità, all'applicativo. Infatti, questi comandi verranno ricevuti dai vari aggregati, i quali rappresentano le entità principali illustrate in precedenza. Questi aggregati permettono la pubblicazione di eventi all'interno dell'Event Store, database che contiene tutti gli eventi passati, da cui leggeranno le varie proiezioni implementate.

Pertanto, dopo aver implementato le configurazioni necessarie al corretto funzionamento di Kafka come ricevitore di messaggi, si è resa necessaria l'implementazione di classi di tipo Parser che potessero ricevere i messaggi ed inviare i relativi comandi agli aggregati.

Si è resa necessaria l'implementazione di classi *'Parser'* poiché, nel momento in cui una centralina installata su un motore di un veicolo ('Dongle') invia un messaggio all'applicativo contenente lo 'stato di salute' del veicolo stesso all'accensione, o allo spegnimento, o nel momento in cui si accende una nuova spia, questo messaggio non viene definito già come comando, ma il relativo comando dovrà essere creato, in modo che possa essere ricevuto in seguito dall'aggregato di riferimento.

Infine, all'interno del dominio di Axon, oltre agli aggregati 'DongleAggregate' ed 'EngineAggregateEntity', sono state implementate le classi per la rappresentazione delle proiezioni e l'Event Store. Tutte queste classi sono alla base di un applicativo basato sugli eventi che intende utilizzare i concetti di Event Sourcing e Command Query Responsibility Segregation.

Una proiezione permette di fornire una vista per il modello di dati sottostante basato su eventi, poiché, nella maggior parte degli applicativi, le richieste di lettura sono di ordine maggiore rispetto alle richieste di scrittura, e, pertanto, risulterebbe

inefficiente dover ricostruire lo stato corrente di un'entità rileggendo tutti gli eventi accaduti ogniqualvolta si riceve una richiesta di lettura.

Sono state realizzate due versioni differenti dello stesso progetto, basate su due tool differenti che permettono di ottenere lo stesso obiettivo.

Per la prima versione si è deciso di sfruttare la possibilità del runtime Axon di assumere una configurazione distribuita basata su Axon Server. Per quest'ultimo è stata utilizzata la versione di tipo standard, disponibile gratuitamente con una licenza di tipo AxonIQ Open Source. Questo server permette di sfruttare le configurazioni di un bus di comandi ed un bus di eventi, che saranno interni alla sua definizione, e non rendono necessarie complesse integrazioni da parte degli sviluppatori.

Per quanto riguarda il database contenete gli eventi, l'Event Store, Axon ne fornisce uno chiamato 'AxonServerEventStore', e permette il collegamento con Axon Server per archiviare e riacquisire eventi.

Come seconda versione dell'applicativo, si è deciso di rimuovere l'utilizzo del Server ed Event Store fornito da Axon Server, poiché il suo utilizzo in un reale applicativo porterebbe a costi aggiuntivi dovuti alla necessità di utilizzare la versione 'Axon Server Enterprise', che aggiunge funzionalità di supporto, monitoraggio, clustering ed altre integrazioni ad Axon Server.

Come Event Store, ne è stato implementato uno basato sul database MongoDB, un repository di tipo NoSQL, la cui principale caratteristica di scalabilità lo rende adatto all'uso di Event Store. Axon fornisce un'iniziale implementazione di 'MongoEventStore', che utilizza MongoDB come database di supporto ed archivia ogni evento in un documento separato. E' possibile modificare questa strategia di archiviazione, poiché Axon fornisce la possibilità di utilizzare come strategia 'DocumentPerCommitStorageStrategy', che crea un unico documento per tutti gli eventi archiviati in un unico commit, cioè nello stesso 'DomainEventStream'. Questo permette di archiviare un unico commit in maniera atomica, però d'altra parte, diventa più complicato interrogare gli eventi manualmente.

Non è necessaria una configurazione complicata per poter utilizzare MongoEventStore, è sufficiente un riferimento a dove poter archiviare le raccolte di eventi nelle proprietà dell'applicativo ed una classe, definita come 'MongoEventStore', al cui interno devono essere introdotte le istanze delle classi 'MongoEventStorageEngine' e 'EmbeddedEventStore' per poter salvare gli eventi all'interno di MongoDb e recuperarli quando necessario.

Inoltre, per la costruzione di un bus di comandi distribuito è stata realizzata un'implementazione di JGroups. Il JGroupsConnector permette di utilizzare JGroups come meccanismo di rilevamento ed invio dei comandi. Dal momento che quest'ultimo gestisce sia il rilevamento dei nodi sia la comunicazione tra essi, JGroupsConnector funge sia da CommandBusConnector sia da CommandRouter.

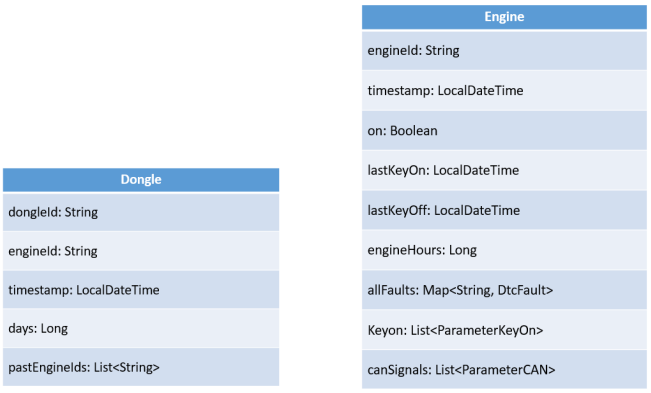


Figura 6.2: Entità definite nell'applicativo

La configurazione dell'implementazione descritta è facilmente realizzata grazie all'utilizzo della dipendenza chiamata 'axon-jgroups-spring-boot-starter'.

6.2 Funzionamento di base

Il progetto in esame prende in considerazione i veicoli attualmente presenti all'interno di un reale applicativo utilizzato dalla società ed il loro stato di salute. Si baserà su due entità, la prima, definita come *'Dongle'*, identifica una centralina installata sul motore di un veicolo, e permette l'invio di diversi tipi di segnali riguardanti lo stato di salute del veicolo. La seconda, definita come *'Engine'*, identifica il motore del veicolo, e pertanto il veicolo stesso.

In relazione alle entità sopra descritte, la realizzazione dell'applicativo è iniziata dalla loro implementazione, in quanto elementi principali del progetto. Queste entità sono descritte in figura 6.2 ed hanno come attributi:

- *Dongle*: identificativo della centralina ed identificativo del motore su cui è installata, ultimo timestamp di ricezione di un messaggio, timestamp in cui è stata installata la centralina sul motore, numero di giorni di installazione della centralina, identificativi passati dei motori su cui era stata installata precedentemente la centralina.

- *Engine*: identificativo del motore, ultimo timestamp di ricezione di un messaggio, ultimo timestamp di accensione e spegnimento del motore, numero di ore in cui il motore è stato acceso, tutti i faults registrati sul motore, tutti i segnali di accensione del motore, tutti i segnali di tipo can.

La centralina installata su un veicolo ('Dongle') permette l'invio di quattro tipi di segnali differenti:

1. segnali *'MasterData'*: segnali che indicano l'installazione (campo 'messageType' uguale a 1) o disinstallazione (campo 'messageType' = 0) di una centraline su un veicolo. (figura 6.3 (a)).

2. segnali *'KeyOn'*: segnali che vengono inviati dalla centralina nel momento dell'accensione del veicolo. Indicano varie informazioni sullo stato attuale di un veicolo. (figura 6.3 (b))

3. segnali *'CAN'*: segnali che vengono inviati ogni 15 minuti dalla centralina presente sul veicolo o allo spegnimento. Come per i segnali di tipo 'KeyOn' indicano varie informazioni sullo stato attuale di un veicolo. (figura 6.3 (c))

4. segnali *'DM1'*: segnali che vengono inviati nel momento in cui si accende una spia riguardante un problema al motore su un veicolo. La spia è univocamente identificata dai campi 'SPN' ed 'FMI', può essere definita come un 'fault', un possibile guasto al motore. (figura 6.3 (d))

Per l'invio dei comandi, il campo 'dongleId' viene creato dall'unione dei campi 'snd' e 'dt', separati da '_'.

In seguito, sono stati definiti gli aggregati relativi ad ogni entità, 'DongleAggregate' ed 'EngineAggregateEntity'. Ogni aggregato può ricevere diversi tipi di comandi, cioè richieste di modifica delle entità, e deve essere in grado di validarli, controllando la loro correttezza e la loro consistenza rispetto allo stato attuale dell'aggregato stesso, ed infine trasformarli in eventi, che verranno salvati all'interno dell'Event Store. E' per questo che, per ogni comando, è stato implementato un gestore di comandi o 'command handler' che permetta la validazione e la successiva pubblicazione del relativo evento. La logica dei comandi verrà descritta nel dettaglio nella sezione 6.6.2.

In seguito, è stata necessaria l'implementazione di tutte quelle proiezioni che intendono ricevere gli eventi pubblicati dagli aggregati. Una proiezione fa parte del lato di lettura dell'applicativo e può essere considerata come una vista per gli eventi pubblicati dagli aggregati.

Ogniqualvolta un nuovo evento viene pubblicato da un aggregato, tutte le proiezioni che contengono un gestore di query relativo a quel comando lo riceveranno, in modo da poter salvare i suoi valori all'interno di un repository dedicato. Infatti, per ogni entità è stata implementata una proiezione, al cui interno sono stati definiti vari gestori di query, o 'query handler', per tutti gli eventi di interesse. La logica della parte di lettura verrà descritta nel dettaglio nella sezione 6.6.3.

In generale, il funzionamento dell'applicativo si basa sulla ricezione di messaggi provenienti dalle varie centraline installate sui veicoli. Ogni centralina invierà un

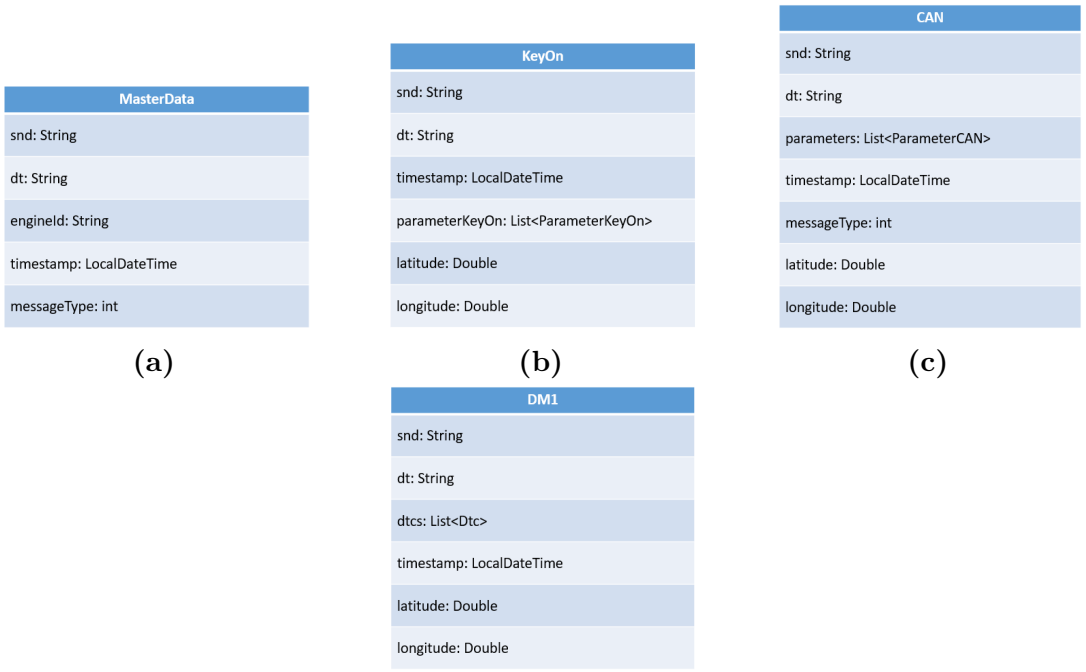


Figura 6.3: Tipologie di segnali provenienti da una centralina installata sul motore di un veicolo

messaggio di accessione del veicolo ad ogni accensione dello stesso, un messaggio di spegnimento ogni volta che verrà spento, ed, inoltre, un messaggio per definire lo stato attuale di un veicolo ogni 15 minuti, ed un messaggio per registrare possibili problemi sul veicolo ogni volta che si accende una nuova spia che possa riguardare problemi al motore.

Questi messaggi sono ricevuti da classi che permettono di analizzarli, e, una volta definito il corretto tipo di messaggio, verrà inviato il relativo comando all'aggregato, in modo da registrare ciò che è accaduto sul veicolo. Infatti, un aggregato che riceve un nuovo comando, si occuperà della sua validazione, ed in seguito della pubblicazione del relativo evento. Gli eventi verranno salvati all'interno dell'Event Store, così che possano essere ripercorsi quando necessario, ad esempio nel caso di ricostruzione dello stato di un'entità o nel caso in cui si voglia aggiungere una nuova proiezione e sia necessario calcolare il suo stato attuale.

Infine, gli eventi verranno anche ricevuti dalle proiezioni implementate, in modo da poter aggiornare il loro stato attuale.

6.3 Metodologie per il Project Management

Per la pianificazione del progetto è stato utilizzato un metodo chiamato *'Event Storming'*. L'Event Storming è un approccio rapido alla modellazione di gruppo per il Domain Driven Design (figura 6.4). Fu introdotto nel 2012 da Alberto Brandolini, e permette di chiarire ciò che accadrà all'interno dell'applicazione, una volta che i singoli processi saranno in esecuzione. Questo tipo di approccio alla modellazione di applicativi, non è una sostituzione del diagramma UML, né tantomeno di altri documenti di progettazione o implementazione, ma piuttosto permette di accelerare e supportare il processo di sviluppo per modellare l'applicativo di interesse ed i suoi processi, con lo scopo finale di avere una comprensione comune a sviluppatori e utenti non tecnici del dominio in cui il software dovrà operare, grazie alla mappatura dei requisiti funzionali del sistema.

Questo metodo si basa su eventi di dominio, definiti come elementi che descrivono come tutto ciò che potrà accadere all'interno del sistema. Una volta scelti questi eventi di dominio, verranno definiti i comandi o trigger, che causano gli eventi, e verranno considerate tutte le possibili origini dei comandi, ad esempio utenti, sistemi esterni o il tempo.

Infine, saranno identificati gli aggregati che accettano i comandi e realizzano gli eventi, e raggruppati in contesti delimitati, a cui si aggiungeranno le possibili relazioni tra contesti differenti, per arrivare a definire una mappa di contesto.

La tecnica dell'Event Storming permette di ridurre il tempo necessario per creare un modello di dominio aziendale completo. Infatti, non è necessario l'utilizzo di un diagramma di tipo UML, ma questo tipo di tecnica permette di scomporre il processo in termini semplici, che possono essere compresi sia da figure tecniche che non, e di esprimere in maniera efficace quello che sarà il comportamento finale del sistema.

6.4 Event Storming in relazione al progetto

In relazione al progetto introdotto, è stata utilizzata la tecnica di Event Storming per mappare i requisiti funzionali del sistema.

In particolare, innanzitutto sono state definite i due aggregati ricorrenti all'interno del sistema in analisi: la centralina e il motore di un veicolo (figura 6.5).

In seguito, sono stati definiti gli eventi di dominio che sarebbero potuti accadere all'interno del sistema, indicati con 'post-it' di colore arancione. Gli eventi di dominio per un veicolo riguardano l'accensione e spegnimento di un veicolo ('Engine On Event', 'Engine Off Event') e la modifica dello stato o dei faults per il motore di un veicolo ('Engine Status Updated Event', 'Engine Faults Updated Event'),

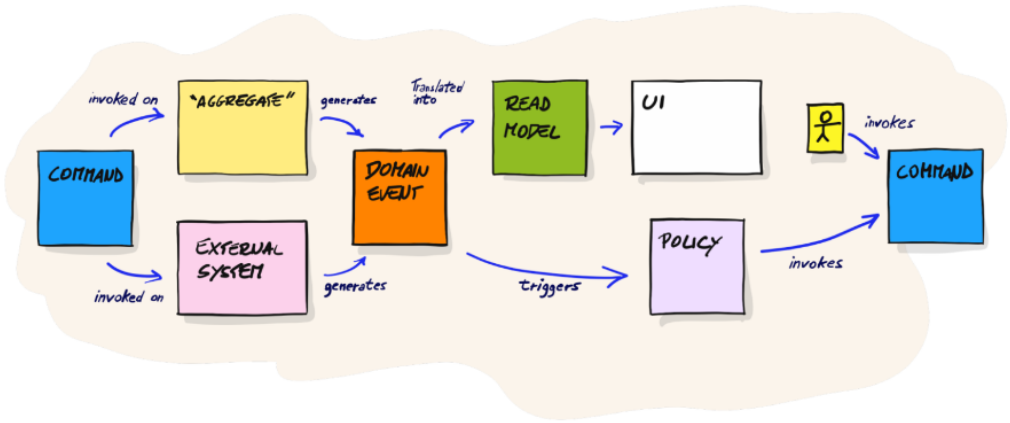


Figura 6.4: Event Storming. Fonte: Introduction to Event Storming, Alberto Brandolini.

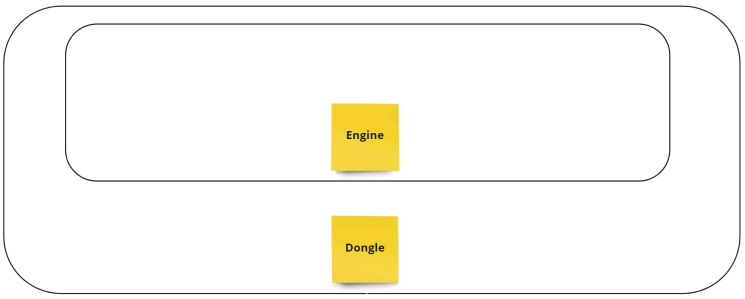


Figura 6.5: Aggregati presenti nel progetto: Engine e Dongle

per una centralina riguardano l'installazione o disinstallazione di una centralina su un veicolo ('Dongle Install Event', 'Dongle Uninstall Event'). Inoltre, sono stati aggiunti come eventi di dominio la ricezione dei messaggi che riguardano i veicoli, che comprendono il messaggio di tipo 'Master data', che descrive gli eventi di installazione o disinstallazione di una centralina, il messaggio di tipo 'Key On', che descrive l'accensione di un veicolo, il messaggio di tipo 'CAN', che descrive l'update dello stato di un veicolo o il suo spegnimento, e il messaggio di tipo 'DM1', che descrive l'update dei faults di un veicolo.

Gli eventi di dominio sono descritti della figura 6.6.

In seguito, sono stati definiti i comandi riguardanti ciascun evento, a partire dai comandi per installazione e disinstallazione di una centralina su un veicolo, definiti come 'Dongle Install Command' e 'Dongle Uninstall Command'. Inoltre gli eventi di 'Engine On' ed 'Engine Off' sono stati affiancati i rispettivi comandi di 'Engine On Command' e 'Engine Off Command', mentre per gli eventi di 'Engine Status Update' ed 'Engine Faults update' sono stati definiti i comandi 'Engine Status

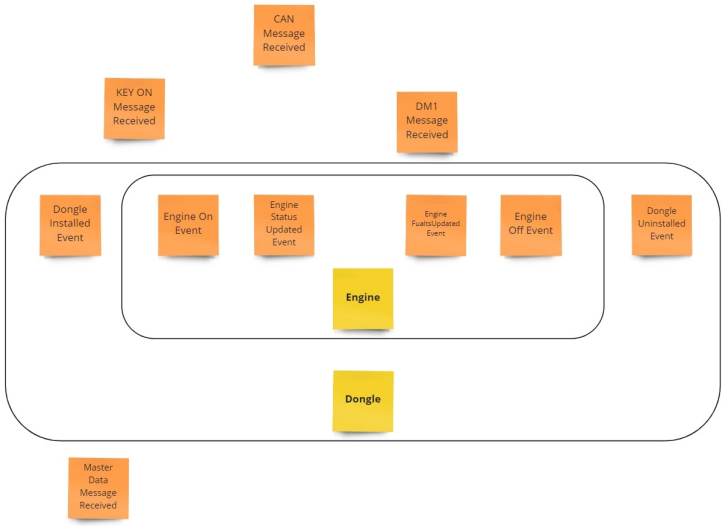


Figura 6.6: Eventi di dominio.

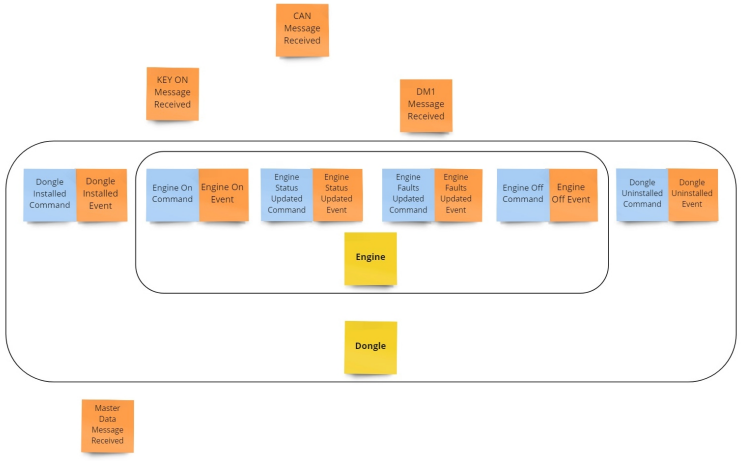


Figura 6.7: Eventi di dominio e relativi comandi.

Update Command' ed 'Engine Faults Update Command'. Tutti i comandi ed i loro rispettivi eventi sono descritti nella figura 6.7.

Non è stato necessario introdurre i comandi per gli eventi che riguardano la ricezione di messaggi di tipo 'Master Data', 'Key On', 'CAN' o 'DM1', poiché questi messaggi vengono ricevuti dall'esterno, e sono utilizzati per la definizione di cambiamenti di stato delle entità prese in considerazione. Infatti, per ognuno di questi messaggi sono state definite delle classi 'parser' (analizzatrici di messaggi), che rappresentano sistemi esterni per analizzare i messaggi e trasformarli in comandi,

inviati agli aggregati, come descritto nella figura 6.8.

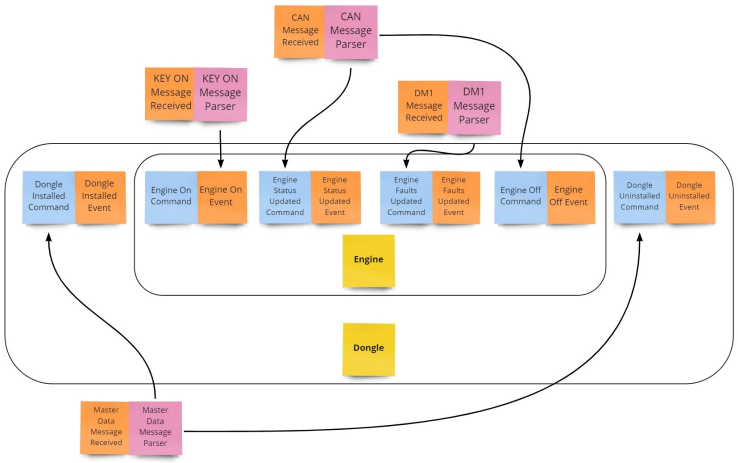


Figura 6.8: Eventi di dominio, relativi comandi e classi 'parser'.

6.5 Strumenti di sviluppo e gestione del software, librerie e framework

Gli strumenti utilizzati per l'implementazione di questo progetto sono strumenti di sviluppo e gestione del software, che permettono di sfruttare al meglio l'utilizzo delle metodologie Event Sourcing e Command Query Responsibility Segregation. In questa sezione non verranno descritti JGroups, Axon Server ed Axon Framework, poiché già descritti in precedenza nei paragrafi 4 e 5.

Java

Java è un linguaggio di programmazione orientato agli oggetti utilizzato per lo sviluppo web. E' stato introdotto nel 1995 da James Gosling per l'azienda Sun Microsystems con il nome di Oak, oggi è di proprietà di Oracle. Questo linguaggio di programmazione è stato progettato per avere il minor numero di dipendenze possibili, ed è per questo motivo che può essere eseguito su molto piattaforme e sistemi operativi differenti. Quando si fa riferimento a Java, è necessario menzionare l'interno ecosistema che cu gravita attorno, composto da Java Virtual Machine (un ambiente di esecuzione virtuale, indipendente dalla piattaforma, che converte il bytecode Java in linguaggio macchina e lo esegue), Java Runtime Environment (un ambiente runtime necessario per l'esecuzione di programmi Java) e Java Development kit (il componente principale dell'ambiente

Java che contiene JRE insieme al compilatore Java, al debugger Java ed altre classi).

Intellij IDEA

Intellij IDEA è un ambiente di sviluppo integrato che supporta differenti linguaggi di programmazione, ad esempio Java o Scala, differenti tecnologie e framework che possono essere utilizzati nel processo di sviluppo software. Fu' sviluppato da JetBrains e pubblicato per la prima volta nel 2001. Fu' il primo IDE ad integrare nuove funzionalità, come la navigazione del codice o il code refactoring.

Gradle

Gradle è uno strumento utilizzato per automatizzare il processo di compilazione di un progetto, dichiarandone le configurazioni in un file separato. Questo tool è basato su un linguaggio specifico di dominio (DSL) basato su Groovy e viene eseguito su Java Virtual Machine. Permette l'utilizzo di diversi repository, come Maven o Ivy, per la gestione delle dipendenze.

Apache Kafka

Apache Kafka è un servizio open source di messaggistica distribuito che permette di pubblicare, sottoscrivere, archiviare ed elaborare flussi di record in tempo reale. Apache kafka fu' introdotto nel 2011 dall'azienda Linkedin, e permette di definire due entità principali, produttore e consumatore. Un produttore ha il compito di pubblicare messaggi su diversi argomenti, mentre un consumatore ha il compito di 'consumare' i messaggi su un determinato argomento. Per ogni argomento, i dati possono essere suddivisi in diverse 'partizioni', all'interno di ognuna delle quali i messaggi sono ordinati sulla base del loro offset (posizione all'interno di una partizione) e indicizzati ed archiviati con un timestamp temporale.

Inoltre, Kafka viene eseguito su un cluster di uno o più server, chiamati 'broker', e le partizioni di diversi argomenti vengono distribuite tra i nodi del cluster e replicate su più broker- Grazie a questo tipo di architettura enormi flussi di messaggi sono pubblicati in modo tollerante agli errori.

Spring framework

Spring è un framework open source per lo sviluppo di applicazioni su piattaforma Java. E' stato creato nel 2002 da Rod Johnson con lo scopo di proporre un'alternativa più "leggera" e flessibile allo standard Enterprise JavaBeans per la realizzazione di applicazioni di livello enterprise.

E' un framework che permette di gestire, tramite 'l'Inversion of control container', l'intero ciclo di vita degli oggetti, detti 'bean', presenti nel contesto applicativo,

a partire dalla loro configurazione, fino al reperimento delle loro dipendenze e la creazione delle singole istanze, il tutto tramite Dependency Injection.

Spring garantisce flessibilità e modularità, poiché i vincoli da rispettare per il suo utilizzo sono minimi, ed elevata testabilità dei singoli componenti di un applicativo.

Spring Boot è una versione del framework Spring che nacque attorno al 2013, quando si richiese la necessità di semplificare alcuni aspetti del framework Spring. In particolare, Spring Boot permette di semplificare la configurazione di un nuovo progetto, poiché tutto ciò che abitualmente viene utilizzato in un'applicazione basata su Spring viene configurato di default, mentre nel caso sia richiesta una qualunque personalizzazione rispetto alle 'impostazioni' di base, questa può essere definita direttamente dallo sviluppatore sfruttando il paradigma di 'convention over configuration'.

Inoltre, poiché Spring Boot viene fornito con un server Tomcat integrato, non è più necessario distribuire l'applicazione su un web server, ma questa può essere compilata in un file di tipo 'jar' che contiene tutte le dipendenze necessarie, incluso il server di applicazioni Tomcat.

Le annotazioni di Spring Boot utilizzate in questo progetto sono:

- @ConfigurationProperties: annotazione utilizzata per iniettare i valori delle proprietà di configurazione in una classe o in un bean.
- @Autowired: annotazione per l'iniezione di bean, consente di non prendere in considerazione la gestione della configurazione dei bean, poiché viene gestita in maniera automatica da Spring.
- @Component: annotazione utilizzata per rilevare e configurare i bean.
- @Repository: annotazione utilizzata per dichiarare che una classe è un repository.

Spring Web

Spring web è un modulo Spring che fornisce supporto per la creazione di applicazioni web. Le annotazioni di Spring Web utilizzate in questo progetto sono:

- @RestController: annotazione che gestisce una richiesta HTTP.
- @RequestMapping: annotazione che indica a Spring a quale tipo di metodo di richiesta HTTP deve essere eseguito il mapping del metodo specificato.

Lombok

Lombok è una libreria che permette di scrivere codice Java in maniera concisa e pulita. Grazie all'utilizzo di Java annotations, la libreria è in grado di generare il codice di getter, setter, construttori, metodi di tipo 'toString', 'equals' ed altri elementi utili.

Tramite l'utilizzo di Java annotation (e reflection) la libreria genererà per noi il codice di getter e setter, costruttori, metodi toString e molti altri elementi utili (e noiosi da definire di volta in volta).

Alcune delle annotazioni utilizzate in questo progetto sono:

- @NoArgsConstructor: annotazione utilizzata per richiedere la creazione di un costruttore vuoto per la classe.
- @AllArgsConstructor: annotazione utilizzata per richiedere la creazione di un costruttore per la classe che utilizzi tutti gli argomenti.
- @Getter: annotazione per richiedere la creazione dei metodi di tipo 'getter' per tutti i campi della classe.
- @Setter: annotazione per richiedere la creazione dei metodi di tipo 'setter' per tutti i campi della classe.
- @Data: annotazione che consente di applicare tutte insieme le annotazioni @ToString, @EqualsAndHashCode, @Getter, @Setter e @RequiredArgsConstructor.

MongoDb

MongoDb, già descritto in relazione alla possibilità di un suo utilizzo come Event Store nel capitolo 4.1, è un database non relazionale orientato ai documenti. E' un database classificato come NoSQL ed è scalabile e permette di ottenere grandi prestazioni. Si differenzia rispetto alla struttura tradizionale basata su tabelle dei database relazionali poiché utilizza un formato JSON, chiamato BSON, per l'archiviazione dei dati.

In relazione al progetto in esame, MongoDB è stato utilizzato come database all'interno del quale salvare le voci relative ad ogni veicolo, in modo che le proiezioni delle entità 'Engine' e 'Dongle' potessero ottenere lo stato attuale di ogni veicolo e di ogni centralina da questi database.

Java Modelling Tools

Java Modelling Tools (JMT) è una suite di applicazioni sviluppate dal Politecnico di Milano e dall'Imperial College of London e rilasciate con licenza GPL, che offre un framewrok completo per poter modellare sistemi con tecniche analitiche o di

simulazione, valutare le prestazioni di sistemi e studiare le capacità e i carichi di lavoro dei sistemi.

Come spiegato nel sito ufficiale della suite [14], l'attuale versione include 6 applicazioni Java:

1. JSIMgraph: Rete di accodamento e simulatore di rete Petri con interfaccia utente grafica.
2. JSIMwiz: Rete di accodamento e simulatore di rete Petri con interfaccia utente basata su procedura guidata.
3. JMVA: Analisi del valore medio e algoritmi di soluzione approssimata per modelli di rete di code.
4. JABA: Analisi asintotica e identificazione dei colli di bottiglia di modelli di reti di code.
5. JWAT: Caratterizzazione del carico di lavoro dai dati di registro.
6. JMCH: simulatore di catena di Markov.

Per questo progetto è stata utilizzata l'applicazione JSIMgraph, che permette la simulazione di una rete con possibilità di accodamento.

6.6 Implementazione dell'applicativo

6.6.1 Aggregati

All'interno di questo progetto sono presenti due aggregati principali:

- Motore di un veicolo ('Engine')
- Centralina installata su un veicolo ('Dongle')

Motore di un veicolo

Il motore di un veicolo rappresenta il veicolo stesso ed è definito con campi principali (figura 6.9):

- engineId: identificativo del motore per ogni veicolo, definito come stringa.
- timestamp: tempo di invio del segnale.
- on: campo definito come booleano che contrassegna se un veicolo è correntemente accesso oppure spento.

- fualts: campo che contiene una lista di elementi di tipo 'DtcStatus', ognuno dei quali definisce un segnale di tipo dtc, il timestamp di arrivo ed il numero di occorrenze.

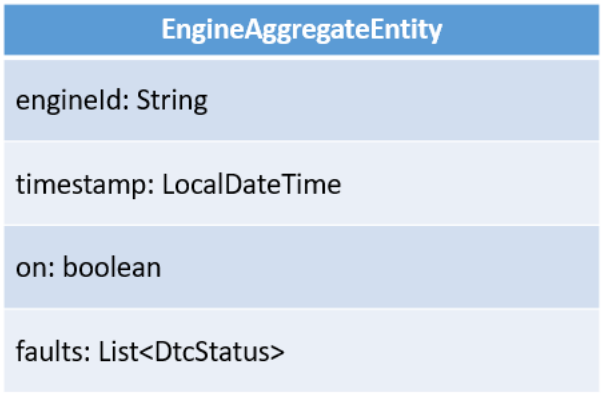


Figura 6.9: Engine Aggregate

Centralina installata su un veicolo

La centralina (Dongle) che viene installata su un veicolo è definita con i seguenti campi (figura 6.10):

- dongleId: identificativo della centralina.
- engineAggregateEntity: campo che identifica su quale motore, e perciò veicolo, è installata la centralina. Questo campo è di tipo EngineAggregateEntity, cioè è l'aggregato di tipo Engine.
- timestamp: tempo di invio del segnale.

6.6.2 Logica dei comandi

La logica dei comandi viene definita tramite l'utilizzo di classi che definiscono la struttura di ogni comando ed i 'command-handler'.

Le classi che descrivono i campi che contraddistinguono ogni comando vengono implementate come strutture dati, ma, nonostante trasportino dati, da sole non hanno particolari funzionalità.

Sono stati implementati i seguenti comandi per l'aggregato Dongle (figura 6.11):

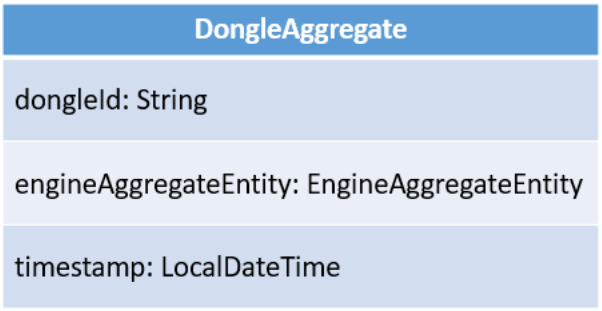


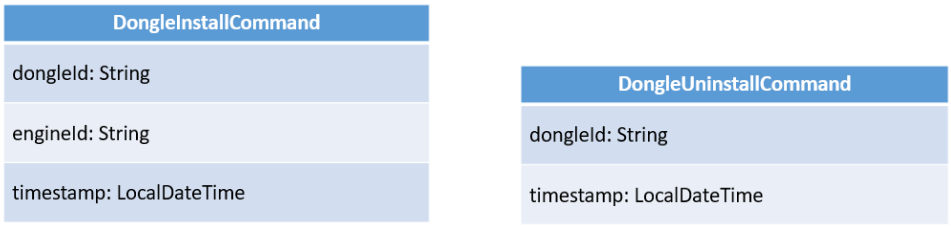
Figura 6.10: Dongle Aggregate

```
@Aggregate
@Component
public class DongleAggregate {
    @AggregateIdentifier
    private String dongleId;
    @AggregateMember
    private EngineAggregateEntity engineAggregateEntity;
    private LocalDateTime timestamp;

    @CommandHandler
    @ValidateOnExecution
    @CreationPolicy(AggregateCreationPolicy.CREATE_IF_MISSING)
    public void handle(@Valid DongleInstallCommand command, Validator validator) {
        Assert.isTrue(validator.validate(command).isEmpty(), message: "error.dongle.validation.failed");
        Assert.isTrue(command.getDongleId().startsWith(DongleUtils.ID_PREFIX), message: "error.dongle.id.format.invalid");
        AggregateLifecycle.apply(new DongleInstalledEvent(
            command.getDongleId(),
            command.getEngineId(),
            command.getTimestamp()
        ));
    }

    @EventSourcingHandler
    public void on(DongleInstalledEvent event) {
        this.dongleId = event.getDongleId();
        this.engineAggregateEntity = new EngineAggregateEntity();
        this.timestamp = event.getTimestamp();
        List<DtcStatus> list = new ArrayList<>();
        this.engineAggregateEntity = new EngineAggregateEntity(event.getEngineId(), event.getTimestamp(), on: false, list);
    }
```

Figura 6.11: Definizione dell'aggregato Dongle in linguaggio Java

- *DongleInstallCommand*: comando che riguarda l'installazione di una centralina su un veicolo. Il comando contiene come campi l'identificativo della centralina (dongleId), l'identificativo del veicolo su cui è stata installata la centralina (engineId) ed il timestamp come identificativo dell'orario di installazione (figura 6.12 (a)).

- *DongleUninstallCommand*: comando che riguarda la disinstallazione di una centralina precedentemente installata su un veicolo. Il comando contiene come campi l'identificativo della centralina disinstallata (dongleId) ed il timestamp per tener traccia del giorno ed orario della disinstallazione (figura 6.12 (b)).



(a) **(b)**

Figura 6.12: Comandi di installazione e disinstallazione di una centralina su un veicolo

Per l'aggregato Engine sono stati implementati i seguenti comandi:

- *EngineOnCommand*: comando che viene inviato durante l'accensione di un veicolo. Questo comando contiene come campi l'identificativo della centralina installata sul veicolo (dongleId), il timestamp che riguarda il momento dell'accensione del veicolo, latitudine, longitudine ed una lista di parametri CAN inviati all'accensione del veicolo, di tipo 'ParameterKeyOn' (figura 6.13 (a)).
- *EngineOffCommand*: comando che viene inviato durante lo spegnimento di un veicolo, contenente come campi l'identificativo della centralina installata sul veicolo (dongleId), il timestamp del momento dello spegnimento del veicolo, latitudine, longitudine ed una lista di parametri CAN inviati allo spegnimento del veicolo, di tipo 'ParameterCAN' (figura 6.13 (b)).
- *EngineStatusUpdateCommand*: comando che viene inviato per aggiornare lo stato di un veicolo. Questo comando contiene come campi l'identificativo della centralina installata sul veicolo (dongleId), il timestamp del momento in cui vengono inviati i valori correnti dei parametri, latitudine, longitudine ed una lista di valori di tipo 'ParameterCAN' che descrivono i parametri CAN (figura 6.13 (c)).
- *EngineFaultsUpdateCommand*: comando che viene inviato per aggiornare i DTC su un veicolo. Contiene come campi l'identificativo della centralina installata sul veicolo (dongleId), il timestamp di invio dei valori per i segnali di tipo 'dtc', latitudine, longitudine ed una lista di valori di tipo 'Dtc' (figura 6.13 (d)).

Per ogni comando deve essere definito come 'TargetAggregateIdentifier' l'identificativo della centralina installata sul veicolo, poiché è necessario che il comando abbia l'identificativo dell'aggregato a cui deve essere recapitato il comando, e dentro il quale si troverà il command handler relativo.

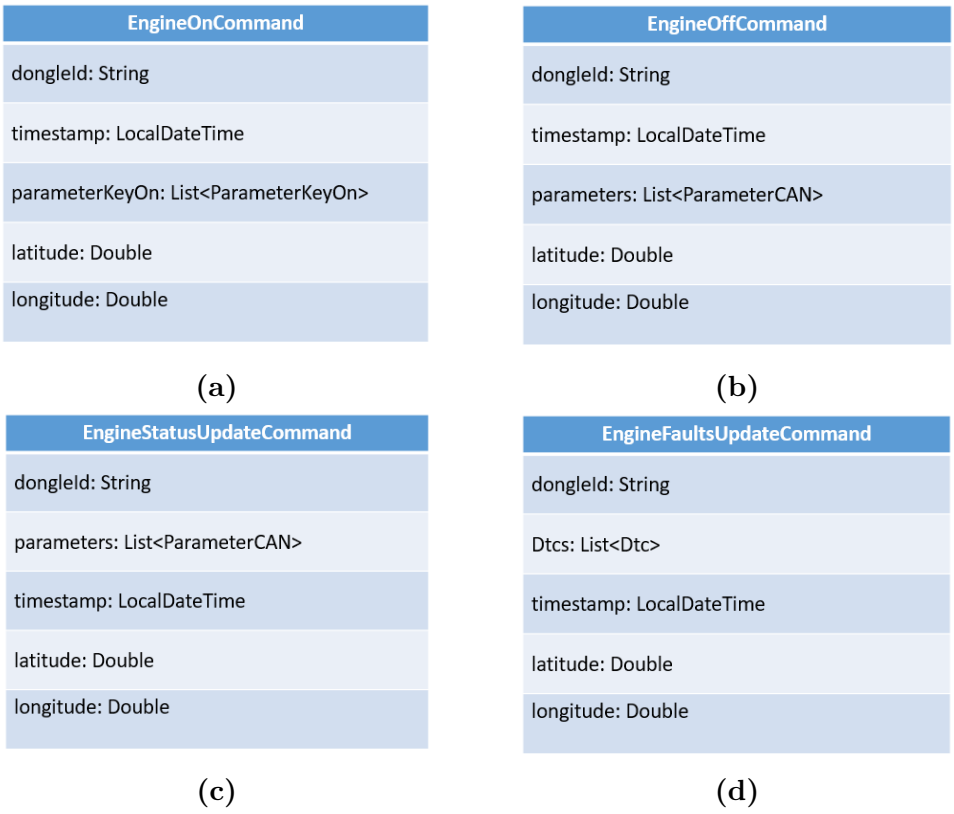
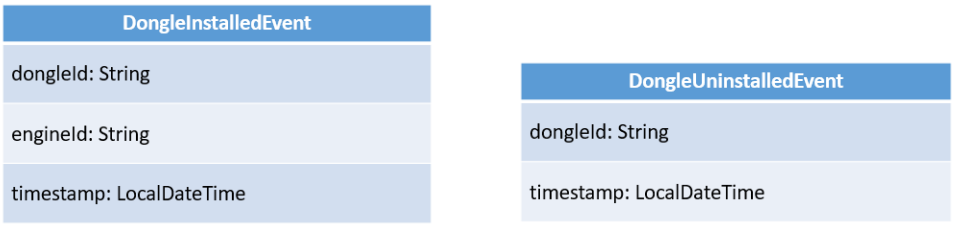


Figura 6.13: Comandi relativi all'aggregato 'Engine'

Un gestore di comandi (o command handler) è responsabile della ricezione dei comandi e dell'archiviazione degli eventi. Dal momento che si rende necessaria la conferma della correttezza di un comando prima che questo venga propagato nel sistema, è fondamentale che avvenga la validazione del comando all'interno di ogni command-handler, controllando che il comando non sia vuoto, che l'identificativo a cui si riferisce sia corretto ed in generale che i suoi campi siano corretti rispetto alle annotazioni definite nella classe che identifica il comando.

Una volta avvenuta la validazione di un comando viene richiamato il metodo 'AggregateLifecycle.apply' che permette di propagare l'evento relativo al comando, perchè questo venga ricevuto dal rispettivo 'EventSourcingHandler' ed i suoi valori vengano quindi salvati nell'Event Store.

All'interno della classe DongleAggregate sono definiti due command-handler, uno per il comando di installazione di una centralina su un veicolo, ed uno per la rispettiva disinstallazione, e due event-sorucing-handler rispettivi, mentre all'interno della classe EngineAggregateEntity sono definiti quattro command-handler, per i comandi di 'EngineOn', 'EngineOff', 'EngineStatusUpdate' ed 'EngineFaultsUpdate',



(a) **(b)**

Figura 6.14: Eventi relativi all'aggregato 'Dongle'

ed i quattro event-sourcing-handler rispettivi.

Dal momento che, per ogni comando corretto che viene ricevuto, è necessaria la pubblicazione di un evento, all'interno di questo progetto per ogni comando è stato implementata una classe che rappresenta l'evento relativo.

In particolare, per quanto riguarda l'aggregato Dongle, sono stati implementati i seguenti eventi:

- *DongleInstalledEvent*: evento relativo all'installazione di una centralina su un veicolo. I campi che costituiscono questo evento sono conformi ai campi presenti all'interno del comando DongleInstallCommand (figura 6.14 (a)).
- *DongleUninstalledEvent*: evento relativo alla disinstallazione di una centralina su un veicolo. I campi che costituiscono questo evento sono conformi ai campi presenti all'interno del comando DongleUninstallCommand (figura 6.14 (b)).

Per quanto riguarda l'aggregato Engine sono stati implementati i seguenti eventi:

- *EngineOnEvent*: evento che riguarda l'accensione di un veicolo. I campi contenuti in questo evento sono conformi a quelli presenti all'interno del comando EngineOnCommand (figura 6.15 (a)).
- *EngineOffEvent*: evento che riguarda lo spegnimento di un veicolo. I campi contenuti in questo evento sono conformi a quelli presenti all'interno del comando EngineOffCommand (figura 6.15 (b)).
- *EngineStatusUpdatedEvent*: evento che viene ricevuto per l'aggiornamento dello stato di un veicolo. I campi contenuti al suo interno sono conformi a quelli presenti sul comando EngineStatusUpdateCommand (figura 6.15 (c)).
- *EngineFaultsUpdatedEvent*: evento che viene ricevuto per l'aggiornamento dei valori DTC su un veicolo. I campi contenuti al suo interno sono conformi a quelli presenti sul comando EngineFaultsUpdateCommand (figura 6.15 (d)).

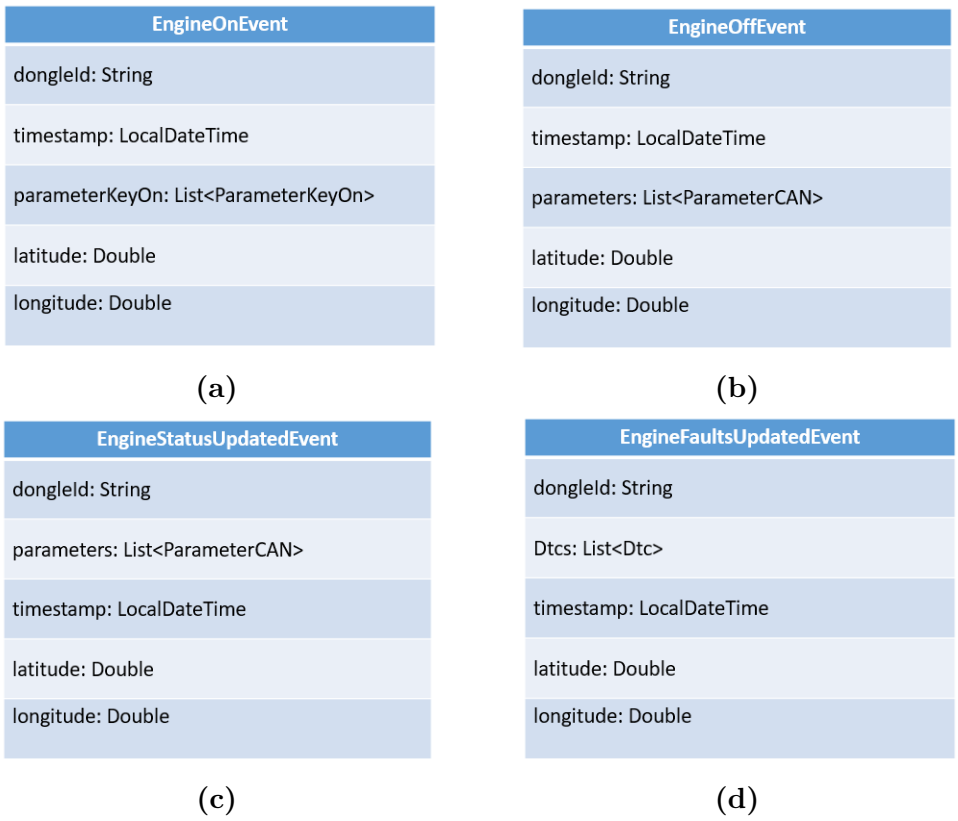


Figura 6.15: Eventi relativi all'aggregato 'Engine'

6.6.3 Logica delle richieste o queries

Non è necessario che il lato query conosca lo stato precedente dell'intero sistema, ma gli è richiesta la conoscenza soltanto dell'attuale rappresentazione dell'entità a cui fa' riferimento.

Per quanto riguarda l'aggregato Dongle, è stata implementata la seguente query:

- *DongleFindByIdQuery*: query per richiedere che venga ritornata l'entità di tipo Dongle, di cui viene specificato il dongleId nella richiesta (figura 6.16 (a)).

Per quanto riguarda l'aggregato Engine, sono state implementate le seguenti query:

- *EngineFindByIdQuery*: query per richiedere che venga ritornata l'entità di tipo Engine, di cui viene specificato l'engineId nella richiesta (figura 6.16 (b)).
- *EngineGetStatusQuery*: query che richiede lo stato corrente (i dtc attualmente attivi) per l'entità di cui vengono specificati l'engineId ed il dongleId nella richiesta (figura 6.16 (c)).

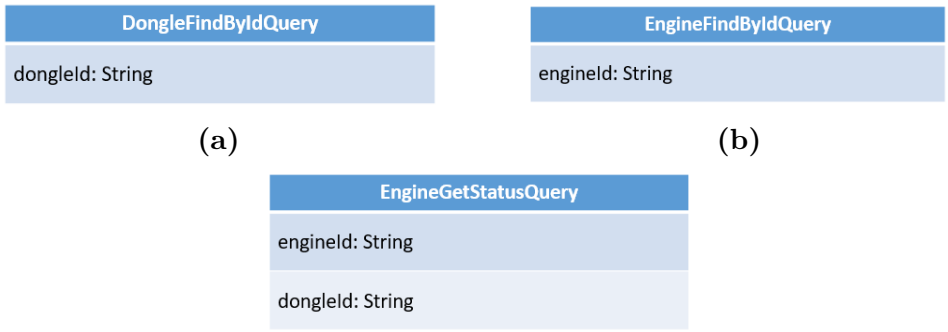


Figura 6.16: Query relative ai due aggregati presi in riferimento, 'Dongle' ed 'Engine'

Inoltre, per ogni aggregato è stata definita la relativa proiezione. Una proiezione (definita anche come 'modello di query') fornisce una vista del modello di dati sottostante basato su eventi, ed è un metodo per derivare il suo stato corrente a partire dal flusso di eventi. La derivazione dello stato corrente di un modello può essere fatta in maniera asincrona rispetto al flusso di eventi, e non è necessario ripercorrere tutto lo storico degli eventi nel momento in cui ne viene creato uno nuovo, bensì basterà mantenere aggiornata la proiezione per una rappresentazione corretta dello stato attuale del sistema.

Per gli aggregati Dongle ed Engine sono state definite le seguenti proiezioni:

- *DongleProjector*
- *EngineProjector*

Per ogni evento, di cui è possibile la pubblicazione all'interno dell'aggregato, esiste un metodo chiamato 'on', che riceve come parametro il metodo stesso, ed annotato con l'annotazione '@EventHandler' che permette di salvare lo stato corrente dell'aggregato e mantenerlo aggiornato nel tempo.

Per ogni query che si intende fare sul sistema, viene dichiarato come per gli eventi un metodo chiamato 'on', ma annotato con l'annotazione ' @QueryHandler', che permette di ritornare una vista aggiornata dello stato attuale dell'entità a cui si fa riferimento.

Entrambe queste proiezioni richiedono come parametro il repository in cui verranno salvati gli stati riguardanti ogni aggregato. I repository standard permettono di memorizzare lo stato effettivo di un aggregato. Per ogni modifica che viene eseguita, lo stato corrente verrà sovrascritto con il nuovo, in modo che le richieste di lettura (query) possano utilizzare le nuove informazioni, usate anche dalla componente di comando.

```java
@Service
public class DongleProjector {

    private final DongleRepository dongleRepository;
    public DongleProjector(DongleRepository dongleRepository){ this.dongleRepository = dongleRepository;}

    @EventHandler
    public void on(DongleInstalledEvent event){
        Optional<Dongle> dongle = this.dongleRepository.findById(event.getDongleId());
        if(dongle.isPresent()) {
            dongle.get().setEngineId(event.getEngineId());
            dongle.get().setTimestamp(event.getTimestamp());
        }
        else {
            Dongle newDongle = new Dongle(
                    event.getDongleId(),
                    event.getEngineId(),
                    event.getTimestamp(),
                    event.getTimestamp(),
                     days: 0L,
                    new ArrayList<>()
                    );
            this.dongleRepository.save(newDongle);
        }
}
```

Figura 6.17: Definizione della proiezione DongleProjector in linguaggio Java

I repository utilizzati dalle entità Dongle ed Engine estende la classe 'MongoRepository' e permette la creazione di database Mongo.

6.6.4 Classi analizzatrici di messaggi

Per poter inviare messaggi di prova, nello stesso modo in cui vengono inviati all'applicativo reale, è stato utilizzato Apache Kafka. In particolare, per l'invio di messaggi dalla classe di test ad una delle classi analizzatrici di messaggi definite è stata utilizzata la classe 'KafkaTemplate', presente nel package 'org.springframework.kafka.core'. Questo template è un wrapper per un produttore Kafka, che si adatta all'interazione con altre funzionalità di Spring, come la 'dependency injection' o la configurazione automatica. Fornisce una serie di metodi pratici per produrre messaggi che saranno consumati da un 'Kafka listener'. E' possibile inviare messaggi con argomenti ('topic') differenti, oppure con partizioni o chiavi differenti.

Per questo progetto sono stati utilizzati quattro topic differenti, relativi ai quattro tipi di messaggi che possono essere ricevuti dall'applicativo:

- *topicMasterData*: riguarda i messaggi di installazione o disinstallazione di una centralina su un veicolo.
- *topicKeyOn*: riguarda il messaggio che segnala l'accensione di un veicolo.
- *topicDm1*: riguarda i messaggi che trasportano segnali di tipo 'DTC'.

- *topicCan*: riguarda i messaggi che trasportano messaggi di tipo 'CAN' o i messaggi che segnalano lo spegnimento di un veicolo.

I messaggi inoltrati nel topic 'topicMasterData' avranno come campi: snd, dt, engineId, timestamp e messageType.

I messaggi inoltrati nel topic 'topicKeyOn' avranno come campi: snd, dt, timestamp, List<ParameterKeyOn> parameterKeyOn, latitude, longitude.

I messaggi inoltrati nel topic 'topicDm1' avranno come campi: snd, dt, timestamp, List<Dtc> dtcs, latitudine, longitudine.

I messaggi inoltrati nel topic 'topicCan' avranno come campi: snd, dt, List<ParameterCAN> parameters,timestamp, messageType, latitudine ,longitudine.

Viene utilizzata l'annotazione '@KafkaListener' per ogni metodo che permette l'ascolto di messaggi inviati dal Kafka producer, all'interno della quale viene dichiarato il topic a cui fa riferimento, in modo tale da permettere al metodo corretto di ricevere i messaggi di un particolare topic.

Questi metodi, dopo aver ricevuto i messaggi relativi al loro topic, utilizzano il 'CommandGateway', che fa parte del package 'org.axonframework.commandhandling.gateway', per inviare i comandi relativi ai commandHandler.

```java
@Service
public class MasterDataParser {

    private static final Logger logger = LoggerFactory.getLogger(MasterDataParser.class);
    private final CommandGateway commandGateway;

    public MasterDataParser(CommandGateway commandGateway) { this.commandGateway = commandGateway; }

    @KafkaListener(topics = "topicMasterData", groupId = "demo")
    void listenDemo(MasterDataMessageReceived event) {
        if(event.getMessageType() == 1) {
            Object resp = commandGateway.send(new DongleInstallCommand(
                    dongleId: DongleUtils.getPrefix() + event.getSnd() + "_" + event.getDt(),
                    event.getEngineId(),
                    event.getTimestamp()
            ));
            logger.debug("Event received from Kafka: {}", event.getClass());
        }
        else {
            Object resp = commandGateway.send(new DongleUninstallCommand(
                    dongleId: DongleUtils.getPrefix() + event.getSnd() + "_" + event.getDt(),
                    event.getTimestamp()
            ));
            logger.debug("Event received from Kafka: {}", event.getClass());
        }
    }
}
```

Figura 6.18: Definizione del parser di messaggi MasterData in linguaggio Java

6.7 MongoEventStore

Poiché nella seconda versione dell'applicazione è stato utilizzato MongoEventStore come Event Store, al posto che 'AxonServerEventStore' fornito da Axon Server per la prima versione dell'applicazione, è necessario spiegare come questo venga implementato.

All'interno dell'applicativo è stata create una classe chiamata 'MongoEventStore', annotata con l'annotazione '@Configuration', all'interno della quale sono presenti due metodi:

- storageEngine: che riceve come parametri la stringa di connessione al database MongoDB ed un serializer di tipo 'Serializer', e permette la costruzione di un 'EventStorageEngine', che verrà utilizzato per la conservazione degli eventi.
- eventStore: che riceve come parametri l'EventStorageEngine appena creato e la configurazione di Axon, e permette di instanziare un 'EmbeddedEventStore che delega l'archiviazione ed il recupero effettivi degli eventi ad un 'EventStorageEngine'.

```
@Configuration
public class MongoEventStore
{

    // The `MongoEventStorageEngine` stores each event in a separate
   MongoDB document
    @Bean
    public EventStorageEngine storageEngine(@Value("${es.mongo.
   connection.string}") String connectionString, Serializer
   serializer)
    {
        return MongoEventStorageEngine
                .builder()
                .mongoTemplate(
                        DefaultMongoTemplate
                                .builder()
                                .mongoDatabase(MongoClients.create(
   connectionString))
                                .build()
                )
                .eventSerializer(serializer)
                .snapshotSerializer(serializer)
                .build();
    }
```

```
    // The Event store 'EmbeddedEventStore' delegates actual storage
    and retrieval of events to an 'EventStorageEngine'.
    @Bean
    public EmbeddedEventStore eventStore(EventStorageEngine
    storageEngine, AxonConfiguration configuration)
    {

        return EmbeddedEventStore.builder()
                .storageEngine(storageEngine)
                .messageMonitor(configuration.messageMonitor(
    EventStore.class, "eventStore"))
                .build();
    }
}
```

6.8 JGroups

L'utilizzo di JGroups come 'DistributedCommandBus' avviene grazie all'utilizzo di configurazioni definite all'interno delle proprietà dell'applicazione.

In particolare, per richiedere ad Axon di non utilizzare Axon server e permettere l'utilizzo di un 'DistributedCommandBus' è necessario specificare le seguenti configurazioni: 'axon.axonserver.enabled=false' ed 'axon.distributed.enabled=true'.

Inoltre, è necessario specificare il file in cui si troveranno le configurazioni di JGroups come 'DistributedCommandBus', all'interno del quale troveremo le seguenti configurazioni:

```
<config xmlns="urn:org:jgroups"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/
    schema/jgroups.xsd">
    <UDP
            mcast_port="${jgroups.udp.mcast_port:45588}"
            ip_ttl="4"
            tos="8"
            ucast_recv_buf_size="5M"
            ucast_send_buf_size="5M"
            mcast_recv_buf_size="5M"
            mcast_send_buf_size="5M"
            max_bundle_size="64K"
            enable_diagnostics="true"
            thread_naming_pattern="cl"
```

```
            thread_pool.min_threads="0"
            thread_pool.max_threads="20"
            thread_pool.keep_alive_time="30000"/>

    <PING />

    <MERGE3 max_interval="30000"
            min_interval="10000"/>
    <FD_SOCK/>
    <FD_ALL/>
    <VERIFY_SUSPECT timeout="1500"  />
    <BARRIER />
    <pbcast.NAKACK2 xmit_interval="500"
                    xmit_table_num_rows="100"
                    xmit_table_msgs_per_row="2000"
                    xmit_table_max_compaction_time="30000"
                    use_mcast_xmit="false"
                    discard_delivered_msgs="true"/>
    <UNICAST3 xmit_interval="500"
              xmit_table_num_rows="100"
              xmit_table_msgs_per_row="2000"
              xmit_table_max_compaction_time="60000"
              conn_expiry_timeout="0"/>
    <pbcast.STABLE desired_avg_gossip="50000"
                   max_bytes="4M"/>
    <pbcast.GMS print_local_addr="true" join_timeout="2000"/>
    <UFC max_credits="2M"
         min_threshold="0.4"/>
    <MFC max_credits="2M"
         min_threshold="0.4"/>
    <FRAG2 frag_size="60K"  />
    <RSVP resend_interval="2000" timeout="10000"/>
    <pbcast.STATE_TRANSFER />
</config>
```

6.9 Comportamento dell'applicativo

Per poter testare l'applicativo, è stato utilizzato Spring Boot, che fornisce un modo conveniente per avviare un contesto applicativo da utilizzare in un test.

Per ogni messaggio che deve essere inviato all'applicativo, è stato creato un metodo con l'annotazione '@Test', all'interno dei quali sono stati creati i maniera casuale i messaggi ed inviati grazie al metodo 'kafkaTemplate.send', a cui sono stati passati il nome del topic di riferimento ed il messaggio da inviare.

Come accennato in precedenza nella sezione 6.6.4, ogni messaggio viene ricevuto dal Kafka Listener in ascolto sul topic relativo. Questo listener utilizza il metodo

'commandGateway.send' per l'invio dei relativi comandi ai Command Handler.

All'interno degli aggregati sono stati implementati i Command Handler, che rimangono in ascolto dei comandi, ed una volta ricevuto un comando, questo viene validato, in modo da controllarne la correttezza prima di essere propagato nell'applicativo, ed in seguito viene richiamato il metodo 'AggregateLifecycle.apply' per la propagazione di un nuovo evento, che conterrà i dati ricevuti dal comando, all'interno del sistema.

Questo evento verrà ricevuto dal relativo Event Sourcing Handler, che, una volta letti i campi necessari, salverà i loro valori come parametri della classe. In questo modo i nuovi attuali valori saranno salvati per l'oggetto.

Inoltre, all'interno di ogni Projection sarà presente un metodo indicato con l'annotazione '@EventHandler', che ascolta la pubblicazione dei relativi eventi e permette di salvare all'interno di repository i nuovi valori delle entità Engine o Dongle.

6.10 Risultati ottenuti

Grazie all'implementazione descritta nel capitolo 6, si ottengono due applicativi, uno che utilizza Axon Server e pertanto 'AxonServerEventStore', l'altro che utilizza MongoDB come Event Store e JGroups come bus di comandi distribuito.

Entrambi gli applicativi mantengono un comportamento conforme, che si differenzia soltanto nel modo in cui avviene il rilevamento dei servizi che intendono ricevere i messaggi. Infatti, nel caso in cui venga utilizzato JGroups come bus di comandi distribuito, è necessario un meccanismo di 'service discovery' che consiste nel realizzare gruppi a cui i vari componenti applicativi possano aderire grazie all'utilizzo di canali creati con lo stesso nome, in modo che questi componenti possano comunicare con i membri del gruppo. Invece, nel caso in cui venga utilizzato Axon Server, Axon sfrutta la cosiddetta 'location transparency', cioè non è richiesto che un componente che invia un messaggio ne conosca la destinazione, ma i messaggi vengono automaticamente instradati ad un componente che abbia dichiarato la possibilità di gestire tali messaggi.

Ogni comando verrà inviato al gestore di comandi di riferimento, che si troverà sull'aggregato corretto. Questo gestore di comandi si occuperà della validazione del comando: è necessari che un comando sia consono rispetto a tutto ciò che è accaduto in precedenza. Ad esempio, nel caso di ricezione di un comando di spegnimento di un veicolo, deve essere verificato che sia stato ricevuto un comando di accensione del veicolo in precedenza e, pertanto, all'interno dell'Event Store deve essere presente un evento di accensione del veicolo preso in considerazione. E' proprio per questo motivo che nel momento in cui un aggregato riceve un nuovo

comando si rende necessaria la ricostruzione dello stato attuale dell'aggregato, facendo richiesta all'Event Store di recuperare tutti gli eventi relativi all'aggregato.

```
1 Assert.isTrue(this.on == true, "error.engine.off");
```

Validazione del comando di 'EngineOff', spegnimento di un veicolo

Una possibile implementazione che può portare ad un'ottimizzazione nei tempi di recupero di grandi moli di eventi sono le tecniche di 'snapshot'. Uno snapshot permette di ottenere una rappresentazione dello stato di un aggregato in un determinato momento, in modo da non dover ripercorrere tutti gli eventi accaduti per ottenere il suo stato attuale, ma soltanto quelli accaduti in seguito rispetto alla creazione dello snapshot.

Una volta che il comando è stato validato e lo stato dell'aggregato ricostruito, verrà creato un nuovo evento relativo al comando ricevuto. Questo evento avrà come valori ciò che è specificato all'interno del comando e verrà pubblicato per tutte le proiezioni in ascolto.

```
AggregateLifecycle.apply(new EngineOffEvent(
            command.getDongleId(),
            command.getParameters(),
            command.getTimestamp(),
            command.getLatitude(),
            command.getLongitude()
    ));
```

Pubblicazione del nuovo evento

L'evento sarà salvato all'interno dell'Event Store, il cui registro degli eventi rappresenta lo stato del database. Gli eventi vengono memorizzati in ordine cronologico, perciò i nuovi eventi verranno aggiunti a seguito degli eventi precedenti. Come già introdotto in precedenza nel capitolo 4.1, nel caso sia necessaria una modifica ad un evento, questo non potrà essere modificato direttamente, ma è inevitabile dover pubblicare un nuovo evento che apporti la modifica richiesta.

Nel momento in cui un gestore di eventi, situato all'interno di una proiezione, riceve un nuovo evento, verrà richiamata l'istanza del repository di riferimento per l'entità presa in considerazione, in modo che i valori all'interno dell'evento possano essere salvati nel database di riferimento.

Ad esempio, una volta ricevuto l'evento di spegnimento di veicolo, si richiederà all'istanza del repository contenente i valori degli eventi ricevuti per i veicolo di trovare l'entry corrispondente al veicolo con il particolare 'dongleId' ricevuto nell'evento. In seguito, si modificherà la entry impostando il valore che dichiara

se un veicolo è acceso o meno a 'false' e impostando anche gli altri valori ricevuti nell'evento, come nell'esempio sottostante.

```
@EventHandler
    public void on(EngineOffEvent event) {
        this.engineRepository.findByCurrentDongleId(event.getDongleId
    ()).ifPresent(engine -> {
                engine.setOn(false);
                engine.setLastKeyOff(event.getTimestamp());
                engine.setEngineHours(engine.getEngineHours() +
    ChronoUnit.MILLIS.between(engine.getTimestamp(), event.
    getTimestamp()));
                engine.setCansignals(event.getParameters());
                engine.setTimestamp(event.getTimestamp());
                engine.setLat(event.getLat());
                engine.setLon(event.getLon());

                this.engineRepository.save(engine);


        });
    }
```

Event handler per evento di 'EngineOff', spegnimento di un veicolo

Nel caso si richieda una lettura di valori salvati all'interno di un database per una relativa entità, come ad esempio si richiede di ottenere lo stato attuale di un veicolo, verrà utilizzato un'istanza della classe 'QueryGateway', che permette di eseguire una query sul sistema.

```
EngineStatusView response = queryGateway.query(new
    EngineGetStatusQuery(
                engineId,
                dongleId
    ), EngineStatusView.class).get();
```

Richiesta dello stato attuale di un veicolo

La richiesta verrà ricevuta dalla proiezione relativa all'entità presa in riferimento ed il repository verrà interrogato per trovare i valori richiesti. In questo caso, una volta ricevuta la richiesta di query, viene interrogato il database per trovare la entry relativa al veicolo preso in riferimento. Nel caso in cui il veicolo sia presente nel database viene ricostruita una lista di 'faults' correntemente attivi sul veicolo ed in seguito creata una nuova istanza della classe 'EngineStatusView' che possa contenere questi valori e ritornata.

```
@QueryHandler
    public EngineStatusView on(EngineGetStatusQuery query){
        Optional<Engine> engine = this.engineRepository.findById(
    query.getEngineId());
        if(engine.isPresent()) {
            List<EngineFaultView> faultsList = new ArrayList<>();

            for(DtcFault fault : engine.get().getAllFaults().values()
    ) {
                if(fault.isCurrentlyOn()) {
                    EngineFaultView faultView = new EngineFaultView(
                            fault.getEcu(),
                            fault.getSpn(),
                            fault.getFmi(),
                            fault.getOccurrenceCount(),
                            fault.getCurrentTimestamp(),
                            fault.getEngineHoursActive()/
    (1000*60*60),
                            fault.getLat(),
                            fault.getLon()
                    );
                    faultsList.add(faultView);
                }
            }
            return new EngineStatusView(
                    engine.get().getEngineId(),
                    engine.get().getCurrentDongleId(),
                    engine.get().getLat(),
                    engine.get().getLon(),
                    engine.get().getEngineHours()/(1000*60*60),
                    faultsList);
        }
        return null;
    }
```

Query Handler per richiesta di ottenimento dello stato attuale di un veicolo

Capitolo 7

Analisi di Benchmark

Dal momento che sono state realizzate due versioni dell'applicativo, per tenere in considerazione sia l'utilizzo di Axon Server, sia la possibilità di utilizzare MongoDB come Event Store e JGroups come bus di comandi distribuito, per ovviare alla necessità dell'utilizzo di una versione di Axon Server a pagamento nel caso di applicativi reali, è presentata in questo capitolo un'analisi di benchmark.

Il benchmarking è un processo di valutazione di una misura rispetto ad uno standard definito in precedenza, che permette di valutare la validità di servizi e processi attraverso le loro performance.

Rispetto al progetto preso in considerazione, si sono resi necessari alcuni test per la misurazione del tempo di risposta generale del sistema e di alcune metriche importanti per un applicativo basato sul Command Query Responsibility Segregation ed Event Sourcing, con o senza l'utilizzo di Axon Server.

Sono noti alcuni problemi dovuti ad applicativi basati su questi concetti:

- Alto numero di eventi: avere un alto numero di eventi porta ad una difficoltà nel mantenerli all'interno dell'Event Store, poiché per il mantenimento dello storico potrebbe essere necessaria una grande quantità di spazio.

- Ricomposizione dello stato di un aggregato: la ricomposizione dello stato di un aggregato, ogni volta che accade una transazione, potrebbe essere un collo di bottiglia per il sistema.

Per poter effettuare un'analisi di benchmarking quanto più veritiera, per prima cosa sono state definite alcune azioni eseguite dall'applicativo a partire dall'invio di un comando fino alla gestione dell'evento relativo, introdotte nella sezione 7.2. In seguito, sono state implementate nuove classi e nuovi metodi, descritti nella sezione 7.3, per poter ottenere i tempi di esecuzione del sistema rispetto ad ogni azione eseguita. Infine è stato utilizzato l'applicativo per simulare un sistema che viene

utilizzato frequentemente ed ottenere dei risultati rispetto alle metriche definite nella sezione 7.1.

7.1 Metriche utilizzate

Per poter effettuare i test di performance dell'applicativo con o senza l'utilizzo di Axon Server, sono state prese in considerazione le seguenti metriche:

- tempo di servizio: tempo necessario per la ricostruzione di uno stato dopo l'invocazione di un comando.
- tempo di risposta del sistema: tempo effettivo da quando un comando viene spedito a quando l'evento relativo viene gestito.
- utilizzazione media: utilizzazione del sistema.
- throughput del sistema: portata media del sistema.
- throughput dei comandi: numero di comandi che riescono a passare all'interno del sistema.
- tempo di attesa: tempo necessario di attesa delle richieste perché possano essere eseguite dal relativo servizio.

7.2 Azioni eseguite dal sistema

Per poter effettuare i test di performance dell'applicativo con o senza l'utilizzo di Axon Server, sono state prese in considerazione tutte le azioni che vengono eseguite dal sistema:

- invio di un comando: momento in cui un comando viene inviato tramite il metodo 'send' o 'sendAndWait' della classe CommandGateway.
- ricezione di un comando prima che venga richiesto il 'fetch' all'Event Store, cioè il recupero di tutti gli eventi che sono accaduti precedentemente sull'aggregato.
- ricezione di un comando da parte del corretto Handler di comandi, dichiarato all'interno di un aggregato con annotazione '@CommandHandler'.
- applicazione di un evento da parte dell'Handler di comandi che ha ricevuti il comando stesso, una volta che questo è stato validato e l'evento inviato.
- ricezione di un evento da parte del corretto Handler di eventi presente nel 'Projector' dell'entità.

- gestione di un evento da parte dell'Handler di eventi, cioè salvataggio all'interno di un repository dei suoi valori.

Queste azioni sono rappresentate da una classe chiamata 'Event', che contiene al suo interno i seguenti parametri:

- timestamp: tempo che definisce quando l'evento è stato creato.

- service: stringa che definisce il servizio che ha creato questo evento.

- eventName: nome dell'evento creato.

- eventId: identificativo dell'evento, creato nel momento della creazione dell'evento.

Questi eventi non sono da confondere con gli eventi che riguardano direttamente l'applicativo creato con Event Sourcing e che verranno salvati nell'Event Store, ma bensì sono necessari per salvare un'azione che viene fatta dal sistema in un preciso momento, e registrare quale servizio la produce.

7.3 Implementazione dell'analisi di benchmark

Prendendo in considerazione tutte le azioni eseguite dal sistema (sezione 7.2) a partire dall'invio di un comando fino alla gestione dell'evento relativo, sono state definite nuove classi classi ed implementati nuovi metodi che saranno introdotti in questa sezione.

Riguardo alla necessità di annotare il momento in cui un comando viene inviato, è stata definita una classe chiamata 'Benchmark', che implementa l'interfaccia 'CommandLineRunner' di Spring Boot. Una volta che il contesto dell'applicazione è stato caricato, Spring Boot chiamerà automaticamente il metodo run, all'interno del quale saranno inviati alcuni comandi nel seguente ordine:

1. un comando di tipo 'EngineOnCommand' per dichiarare che il veicolo con un definito 'dongleId' è stato acceso.

2. mille comandi di tipo 'EngineStatusUpdateCommand' che permettono di dichiarare la modifica nello stato di un veicolo.

3. un comando di tipo 'EngineOffCommand' per dichiarare che il veicolo con lo stesso 'dongleId' è stato spento.

Per poter prendere in considerazione il tempo trascorso da quando il comando è stato ricevuto dal sistema, ma il 'fetch' di tutti gli eventi riguardanti l'aggregato

non è ancora stato richiesto, ed il momento in cui il 'fetch' è terminato, è necessaria l'introduzione di una nuova classe, chiamata 'MyCommandHandlerInterceptor', che permette di intercettare i messaggi prima che venga richiesto il 'fetch' nell'Event Store. Questa nuova classe implementa l'interfaccia MessageHandlerInterceptor, che dichiara un metodo chiamato 'handle' che accetta due parametri: l'attuale Unit Of Work ed un InterceptorChain. Quest'ultimo viene utilizzato per permette la continuazione del processo di invio di un messaggio grazie al metodo chiamato 'proceed'. L'unit of Work, invece, permette di ottenere il messaggio che si sta processando.

Dal momento che questo MessageHandlerInterceptor deve essere registrato nel Command Bus perchè possa intercettare tutti i messaggi, nell'applicativo che utilizza Axon Server è stata definita una classe di configurazione 'CommandBusConfiguration', annotata con '@Configuration', che registra l'intercettore dei messaggi grazie al metodo 'registerHandlerInterceptor' presente all'interno della classe 'CommandBus', che rappresenta il bus dei comandi correntemente utilizzato.

Per quanto riguarda l'applicativo che utilizza JGroups come bus di comandi distribuito e MongoDB come Event Store, è stata definita una classe denominata 'FixReceiver' che contiene al suo interno un metodo chiamato 'jgroupsConnectorFactoryBean'. All'interno di questo metodo viene dichiarata una variabile di tipo 'JGroupsConnectorFactoryBean', un Bean di Spring che crea un JGroupsConnector e lo avvia all'avvio del contesto dell'applicazione. Per questa classe viene richiamato il metodo 'setChannelFactory', all'interno del quale viene creato un nuovo 'JChannel' e dentro il quale si trova il metodo 'toMyMessage' che può essere usato per l'intercettazione die messaggi.

Per ogni azione riportata in precedenza di cui è richiesto il salvataggio per permettere la verifica delle performance, viene richiamato il metodo 'logEvent' della classe 'EventLogger', che permette il salvataggio di un evento all'interno di una lista di eventi di tipo 'Event'.

La classe 'EventLogger' implementa l'interfaccia 'DisposableBean', che viene implementata dai Bean che vogliono rilasciare le risorse in caso di distruzione. Al suo interno è contenuto il metodo 'destroy' per la distruzione del Bean stesso, in questo caso utilizzato per scrivere la lista completa di eventi di tipo 'Event' su un file di tipo 'csv' una volta che l'applicazione è terminata.

7.4 Funzionamento dell'analisi di benchmark

Nel momento in cui viene inviato un comando all'interno del metodo 'run' della classe 'Benchmark' viene richiamato il metodo logEvent per registrare l'invio di un comando ad un Handler di comandi:

```
String eventId = UUID.randomUUID().toString();
eventLogger.logEvent(this.getClass(), new Event(LocalDateTime.now(),
    this.getClass(), "CommandSent", eventId));
```

Una volta che il comando inviato viene intercettato dall'intercettore dei messaggi dichiarato, è necessario per prima cosa controllare che il messaggio sia effettivamente quello atteso, poiché l'intercettore dei messaggi non è in grado di fare distinzione tra i messaggi per ricevere soltanto quelli di nostro interesse. In seguito verrà registrato un nuovo evento poiché il comando è stato ricevuto prima della richiesta di 'fetch' di tutti gli eventi accaduti per l'aggregato preso in considerazione all'Event Store.

```
var msg = unitOfWork.getMessage();
    if(msg.getPayload().toString().contains("
   EngineStatusUpdateCommand")){
        var evtTrackingId = ((EngineStatusUpdateCommand)      msg.
   getPayload()).getEvtTrackingId();
        EventLogger.lastEvent = evtTrackingId;
        EventLogger.logEvent(new Event(LocalDateTime.now(), "
   MessageHandler", "CommandReceivedBeforeFetch", evtTrackingId));
        }
```

In seguito, è stato posto il salvataggio dell'evento a seguito della richiesta di 'fetch' all'interno dell'Handler dell'evento 'DongleInstalledEvent':

```
EventLogger.logEvent(this.getClass(), new Event(LocalDateTime.now(),
    this.getClass(), "CommandReceivedAfterFetch", EventLogger.
    lastEvent));
```

A questo punto il comando viene ricevuto dal corretto Handler di comando, all'interno del quale è stato posto il salvataggio dell'evento ricevuto:

```
EventLogger.logEvent(this.getClass(), new Event(LocalDateTime.now(),
    this.getClass(), "CommandReceived", command.getEvtTrackingId()));
```

Infine, sempre all'interno dell'Handler corretto del comando viene richiamato il metodo 'AggregateLifecycle.apply' per pubblicare l'evento relativo al comando comando ricevuto e perciò sarà nuovamente richiamato il metodo 'logEvent':

```
EventLogger.logEvent(this.getClass(), new Event(LocalDateTime.now(),
    this.getClass(), "EventApplied", command.getEvtTrackingId()));
```

E come ultima azione, l'evento pubblicato dall'Handler di comandi viene ricevuto dalla proiezione all'interno del metodo Handler di evento, perciò troveremo:

```
EventLogger.logEvent(this.getClass(), new Event(LocalDateTime.now(),
    this.getClass(), "EventReceived", event.getEventTrackingId()));
```

per registrare la ricezione di un evento, e:

```
EventLogger.logEvent(this.getClass(), new Event(LocalDateTime.now(),
    this.getClass(), "EventHandled", event.getEventTrackingId()));
```

per annotare la corretta gestione dell'evento.

7.5 Utilizzo di Java Modelling Tools

Java Modelling Tools è stato utilizzato per implementare un sistema che potesse risultare simile ad un reale sistema basato su Event Sourcing e Command Query Responsibility Segregation.

Per questo motivo sono state introdotte nella simulazione alcune 'stazioni' che rappresentano le diverse entità presenti nell'applicativo. Per ogni stazione è necessario specificare la media e la deviazione standard della tempistica necessaria per la corretta gestione delle sue attività.

In particolare, come introdotto nella figura 7.1, sono state definite 8 stazioni:

- *Source*: stazione sorgente, posta come inizio della simulazione.
- *JGroups*: rappresenta il bus di comandi distribuito.
- *AggregateService*: rappresenta un aggregato, all'interno del quale sarà presente il Command Handler relativo al comando preso in esame.

  Si terrà in considerazione il tempo trascorso da quando il comando è stato ricevuto, una volta che sia già stata completata la fase di 'fetch' dello storico dell'aggregato, ('CommandReceivedAfterFetch') a quando il comando è ricevuto dall'Handler corretto ('CommandReceived').
- *EventStore*: Rappresenta l'Event Store presente all'interno dell'applicativo. Si terrà in considerazione il tempo trascorso da quando il comando viene intercettato prima della fase di 'fetch' dello storico dell'aggregato ('CommandReceivedBeforeFetch') a quando è stata terminata la fase di 'fetch' ('CommandReceivedAfterFetch'). In questo modo si può definire l'impatto che ha l'Event Store sulle prestazioni per quanto riguarda la richiesta di recupero dello storico di un aggregato ogni volta che viene inviato un nuovo comando.

- *ClassSwitch 1*: permette di tenere in considerazione quei comandi che non vengono validati e, pertanto, non possono arrivare alla pubblicazione del relativo evento da parte dell'Event Store Mongo.
- *MongoBus*: Rappresenta l'Event Store all'interno del quale verranno salvati tutti gli eventi pubblicati da diversi gestori di comandi. Si terrà in considerazione il tempo trascorso da quando un evento viene pubblicato da un Handler di comandi ('EventApplied') a quando questo viene ricevuto dalla proiezione corretta ('EventReceived').
- *Projection*: rappresenta la proiezione, all'interno della quale sarà presente l'Event Handler per l'evento relativo al comando preso in esame.

  Si terrà in considerazione il tempo trascorso da quando l'evento è stato ricevuto ('EventReceived') a quando l'evento è stato gestito ('EventHandled').
- *Sink 1*: stazione terminale, posta come fine della simulazione.

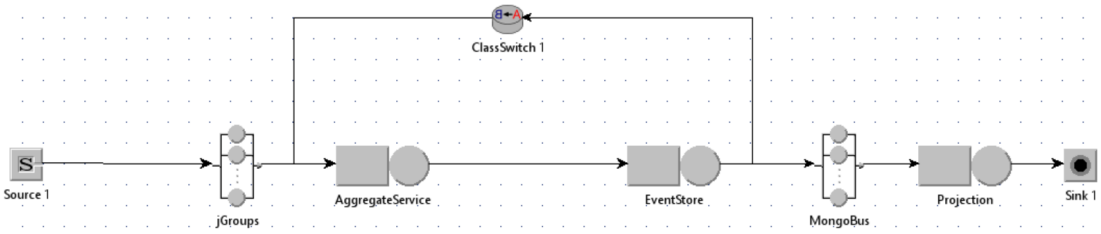


Figura 7.1: Progetto utilizzato per l'applicativo Java Modelling Tools

7.6 Risultati ottenuti

Dal momento che per ogni stazione è necessario specificare la media e la deviazione standard delle tempistiche necessarie per il completamente delle attività, innanzitutto è stato testato per tre volte l'applicativo di riferimento, inviando 1000 comandi alla volta di tipo 'EngineStatusUpdateCommand' e, per ogni run, sono stati salvati i tempi impiegati dall'applicativo necessari per il completamento di ogni azione definita in precedenza nella sezione 7.2. In seguito, sono state calcolate la media e la deviazione standard delle tre run, in modo da utilizzare i tempi calcolati come parametri delle stazioni della simulazione.

Applicativo senza Axon Server

Per quanto riguarda l'applicativo che fa uso di JGroups come bus di comandi distribuito e MongoDb come Event Store, sono stati ottenuti i risultati esposti nella tabella 7.1.

Azioni	Average	Standard Deviation
CommandSent - CommandReceivedBeforeFetch	1.296	1.3455117
CommandReceivedBeforeFetch - CommandReceivedAfterFetch	36.66733	22.7210603
CommandReceivedAfterFetch - CommandReceived	52.072926	33.6728574
CommandReceived - EventApplied	0.14	0.447660
EventApplied - EventReceived	596.933	259.3272697
EventReceived - EventHandled	5.7	4.3911361

Tabella 7.1: Tempistiche per applicativo senza l'utilizzo di Axon Server

Grazie alla simulazione effettuata con Java Modelling Tools, in cui il massimo numero di campioni che il sistema poteva ricevere è stato impostato ad un milione, si è potuto osservare che i principali colli di bottiglia sono dovuti ai servizi AggregateService ed EventStore. Infatti, in seguito ad una simulazione delle prestazioni del sistema si può osservare una difficoltà nel soddisfare le richieste da parte di questi servizi, poiché i tempi di residenza delle richieste al loro interno sono molto alti, così come il tempo di risposta per ogni richiesta. Infatti, mentre l'utilizzazione media raggiungeva il 100% nel giro di pochi millisecondi, il throughput massimo rimaneva attorno allo 0.02 a millisecondo.

Per questo motivo, si è deciso di aumentare le repliche dei servizi in modo da ottenere un tempo medio di gestione delle richieste superiore. Il numero di repliche per ogni servizio, l'utilizzazione media ed il throughput attuale sono descritti nella tabella 7.2.

In questo modo la lunghezza media dei messaggi presenti nella coda di attesa dei servizi può diminuire, poiché saranno soddisfatte più richieste al millisecondo.

Servizio	Numero di repliche	Utilizzazione media	Throughput
AggregateService	50	83.8%	1.07
EventStore	40	41%	0.353
Projection	4	52.4%	-

Tabella 7.2: Numero di repliche minime per applicativo senza l'utilizzo di Axon Server

E' necessario tenere in considerazione che il servizio 'Projection', cioè la proiezione che si occupa del salvataggio dei valori ricevuti da un evento pubblicato, potrebbe avere performance poco soddisfacenti ed essere considerato un 'collo di bottiglia', ma non è da considerarsi gravemente impattante nelle performance dell'intero sistema grazie al principio della 'location transparency', che sostiene che i consumatori di un servizio non conoscano la posizione di un servizio finché questo non viene individuato nel registro. Poiché il servizio viene ricercato ed associato in

maniera dinamica, è possibile che la sua implementazione si sposti da una posizione all'altra all'insaputa del client.

Il *throughput medio* del sistema, cioè la sua capacità effettiva di trasmissione, è di circa 0.35 messaggi al millisecondo, il che significa che 3.5 messaggi entrano nel sistema e vengono gestiti ogni 10 millisecondi (figura 7.2), il *tempo medio di risposta* del sistema è di circa 775 millisecondi (figura 7.3).

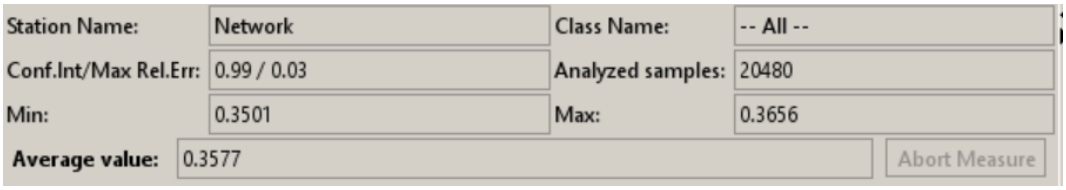


Figura 7.2: Throughput del sistema

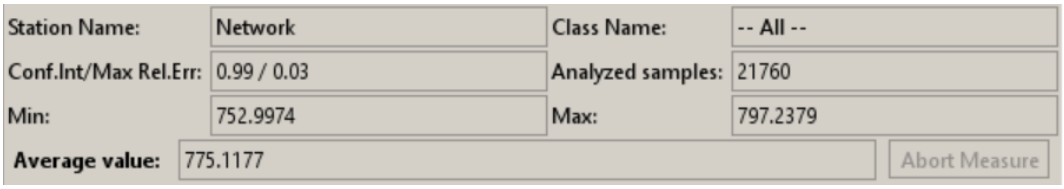


Figura 7.3: Tempo medio di risposta del sistema

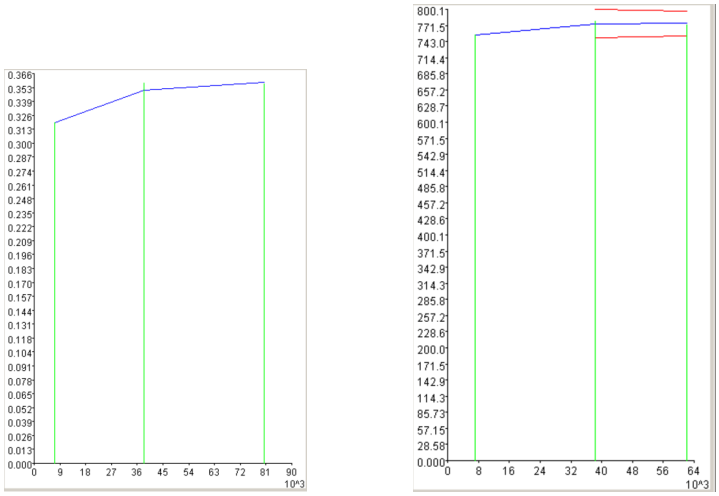
Applicativo con Axon Server

Per quanto riguarda l'applicativo che fa uso di Axon Server, si può notare nella tabella 7.3 un sostanziale decremento nei tempi medi necessari per compiere le azioni descritte nella sezione 7.2.

Come per l'applicativo che non fa uso di Axon Server, i principali colli di bottiglia sono i servizi 'AggregateService' ed 'EventStore'. Infatti, come in precedenza, si può osservare una difficoltà nel soddisfare le richieste da parte di questi servizi, per cui l'utilizzazione raggiungeva il 100% nel giro di pochi millisecondi, il throughput rimaneva molto basso, mentre il tempo di attesa in coda dei messaggi ancora da gestire aumentava in maniera esponenziale.

Per poter fare un confronto con la precedente versione dell'applicativo, sono state utilizzate le stesse quantità di repliche dei servizi e si sono potuti osservano i risultati riportati nella tabella 7.4.

Il *throughput medio* del sistema basato su Axon Framework e Axon Server è circa 0.36 messaggi al millisecondo, che significa che circa 3.6 messaggi entrano nel sistema e vengono gestiti ogni 10 millisecondi (figura 7.5), mentre il *tempo medio di risposta* del sistema è di circa 164 millisecondi (figura 7.6).



(a) **(b)**

Figura 7.4: Grafici di throughput e tempo medio di risposta del sistema per l'applicativo che non utilizza Axon Server

Azioni	Average	Standard Deviation
CommandSent - CommandReceivedBeforeFetch	0.047	0.4866117
CommandReceivedBeforeFetch - CommandReceivedAfterFetch	11.46953	8.9385384
CommandReceivedAfterFetch - CommandReceived	48.81818	28.8565208
CommandReceived - EventApplied	0.107	1.0805396
EventApplied - EventReceived	13.717	23.7927773
EventReceived - EventHandled	8.073	5.5668249

Tabella 7.3: Tempistiche per l'applicativo con l'utilizzo Axon Server

Servizio	Numero di repliche	Utilizzazione media	Throughput
AggregateService	50	72.9%	1.83
EventStore	40	35%	0.38
Projection	4	67.01%	-

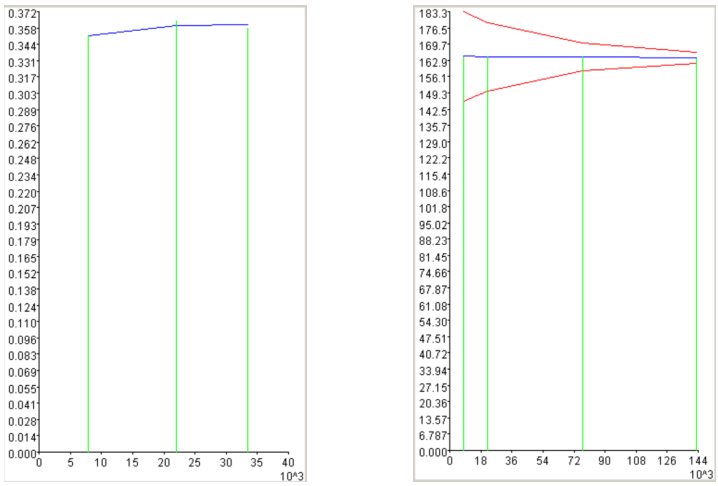
Tabella 7.4: Numero di repliche minime per l'applicativo con l'utilizzo di Axon Server

Station Name:	Network	Class Name:	-- All --
Conf.Int/Max Rel.Err:	0.99 / 0.03	Analyzed samples:	9600
Min:	0.3509	Max:	0.3720
Average value:	0.3612		Abort Measure

Figura 7.5: Throughput del sistema

Station Name:	Network	Class Name:	-- All --
Conf.Int/Max Rel.Err:	0.99 / 0.03	Analyzed samples:	51000
Min:	161.7549	Max:	166.3401
Average value:	164.0475		Abort Measure

Figura 7.6: Tempo medio di risposta del sistema



(a) **(b)**

Figura 7.7: Grafici di throughput e tempo medio di risposta del sistema per l'applicativo con Axon Server

Capitolo 8

Conclusioni

L'obiettivo principale di questa tesi è stato quello di implementare e confrontare alcune tecniche volte alla creazione di un'applicazione orientata agli eventi, al fine di aiutare gli sviluppatori nel scegliere il modo migliore per implementare nella pratica i concetti di Event Sourcing e Command Query Responsibility Segregation.

In particolare, sono state utilizzate due implementazioni differenti che possano soddisfare esigenze diverse quando si intende sviluppare un applicativo orientato agli eventi. Entrambe le implementazioni si basano sull'utilizzo del framework chiamato 'Axon', che permette di creare microservizi evoluti orientati agli eventi, ed è basato sui principi di Domain Driven Design, Command Query Responsibility Segregation ed Event Sourcing. Questo framework, nonostante sia ancora poco conosciuto, fornisce vari componenti pronti all'uso per l'applicazione dei principi descritti in precedenza, come ad esempio aggregati, bus di comandi, eventi e query.

La differenza principale nelle due implementazioni dello stesso applicativo, introdotte nel capitolo 6, consiste nell'utilizzo o meno di Axon Server.

Axon Server, rilasciato sotto licenza AxonIQ, è un server stand-alone che rende disponibile l'utilizzo di un Event Store, database per il salvataggio degli eventi prodotti dall'applicazione, introdotto nel capitolo 4.1, e un router di eventi. Questo server permette di elaborare e mantenere milioni di eventi senza particolari problemi, ed, inoltre, si occupa dell'inoltro dei messaggi ai vari applicativi senza necessità di alcuna configurazione.

Dal momento che la versione standard gratuita di Axon Server potrebbe non essere sufficiente, in termini di prestazioni, per un reale applicativo, si renderebbe necessario l'utilizzo della versione a pagamento 'Axon Server Enterprise', che porta a costi aggiuntivi nel mantenimento dell'applicativo. E' per questo motivo che, all'interno di questo documento, è stata presentata una seconda implementazione dell'applicazione che non fa uso di Axon Server, ma si basa sull'utilizzo di MongoDb come Event Store e JGroups per la costruzione di un bus di comandi distribuito.

Per quanto riguarda la prima versione dell'applicativo introdotto in questo documento, cioè la versione che utilizza Axon Server, ne è risaltata la facilità di implementazione dovuta all'utilizzo di questo server, poiché, nonostante possa sembrare complessa la gestione delle configurazioni all'interno di un microservizio, come la corretta impostazione del rilevamento dei servizi o l'instradamento degli eventi, tutto questo viene gestito in automatico dal server, senza necessità di alcuna configurazione da parte dello sviluppatore. Inoltre, nel caso si richieda un ridimensionamento orizzontale dell'applicativo, è sufficiente la creazione di nuove istanze di un servizio, che verranno gestite correttamente dal server. Infine, grazie ad un'implementazione intrinseca dell'Event Store, offre caratteristiche di scalabilità e throughput superiori rispetto ad un database relazionale.

Riguardo alla seconda versione dell'applicativo, che non richiede l'utilizzo di Axon Server, si è resa necessaria una maggior implementazione dei vari componenti che la costituiscono, poiché non sono più disponibili le implementazioni di Event Store e bus di comandi dovute ad Axon Server. Infatti, come Event Store è stato utilizzato MongoDb e come bus di comandi distribuito è stato utilizzato JGroups. Perciò, nel caso si scelga di non utilizzare il servizio proposto da AxonIQ, è necessaria una conoscenza approfondita del funzionamento di un applicativo orientato agli eventi ed anche del database MongoDb e della libreria JGroups.

Indubbiamente, per un'azienda che intende creare un applicativo basato sugli eventi, questa seconda implementazione permette di ridurre i costi dovuti al suo funzionamento e mantenimento, poiché, come già illustrato in precedenza, non è necessaria alcuna licenza a pagamento.

Come presentato nel capitolo 7 che illustra l'analisi di benchmark condotta per sottolineare la differenza nelle due implementazioni proposte, si possono notare alcune differenze negli applicativi.

In primo luogo, nonostante il throughput medio del sistema, a parità del numero di repliche dei servizi, sia similare, 3.5 messaggi al secondo per l'applicativo senza l'utilizzo di Axon Server e 3.6 messaggi al secondo per l'applicativo che ne fa uso, il tempo medio di risposta del sistema è indiscutibilmente più alto per il primo applicativo, il ché indica una difficoltà maggiore di processamento ed instradamento dei messaggi portata dalla mancanza di un'automatizzazione di queste attività.

Ciò che potrebbe essere un 'collo di bottiglia' per entrambe gli applicativi è il servizio di proiezione, 'Projection', che si occupa del salvataggio dei valori ricevuti dagli eventi pubblicati. Questo servizio non è stato preso in considerazione nell'analisi delle prestazioni del sistema poiché, a sostegno del principio della 'location transparency', il servizio viene ricercato ed associato in maniera dinamica.

Inoltre, l'utilizzazione media ed il throughput di ogni servizio, descritti nelle tabelle 7.2 e 7.4, sono generalmente più bassi nell'applicativo che utilizza Axon Server rispetto all'applicativo che non ne fa uso.

Pertanto, grazie allo studio ed implementazione di questi due applicativi, si è

potuto comprendere come un'applicazione orientata agli eventi potrebbe beneficiare ampiamente dell'utilizzo del server messo a disposizione da AxonIQ, nonostante questo possa portare a dei costi aggiunti dovuti alla sua licenza. Infatti, oltre a diminuire notevolmente la quantità di implementazione necessaria nel creare l'applicativo, porta ad un miglioramento delle sue prestazioni e tempi di risposta del sistema.

Grazie a questo elaborato si è potuto comprendere quanto un'applicazione orientata agli eventi, che faccia uso di Axon Framework ed Axon Server, possa essere utile nel mondo dei microservizi. Dal momento in cui si verifica un nuovo evento, questo può essere utilizzato per guidare le decisioni su tutto ciò che è in attesa del verificarsi di quell'evento. Ogni servizio è completamente scollegato dagli altri, poiché colui che pubblica un nuovo evento non chiamerà più colui che riceverà l'evento, ma agirà su di esso in modo eventualmente coerente.

In conclusione, quindi, i risultati mostrano come l'integrazione dei principi di Event Sourcing e Command Query Responsibility Segregation e l'utilizzo di Axon Framework ed Axon Server possano portare a grandi benefici per tutti gli applicativi che possano essere trasformati in applicazioni orientate agli eventi.

Bibliografia

[1] Greg Young. *Cqrs, Event Sourcing, and Ddd: Finding Simplicity in Complex Systems*. Addison-Wesley Professional, 2020 (cit. alle pp. 2, 16).

[2] Martin Fowler. *Event Sourcing*. URL: `https://martinfowler.com/eaaDev/EventSourcing.html` (cit. alle pp. 2, 10).

[3] *Reactive Manifesto*. URL: `https://www.reactivemanifesto.org/it` (cit. a p. 3).

[4] *What is SOA, or service-oriented architecture?* URL: `https://www.ibm.com/cloud/learn/soa` (cit. a p. 6).

[5] Richards Mark e Ford Neal. *Fundamentals of Software Architecture*. O'Reilly Media, Inc., 2020 (cit. a p. 8).

[6] *Terminologia del Domain Driven Design*. URL: `https://www.dddcommunity.org/resources/ddd_terms/` (cit. a p. 12).

[7] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988 (cit. a p. 16).

[8] Martin Fowler. *CQRS*. URL: `https://martinfowler.com/bliki/CQRS.html` (cit. a p. 16).

[9] Deepshi Garg. *CAP Theorem*. URL: `https://medium.com/@deepshig/cap-theorem-e52bfc855b75` (cit. a p. 19).

[10] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003 (cit. alle pp. 20, 21).

[11] Martin Fowler. *Schemaless Data Structures*. URL: `https://martinfowler.com/articles/schemaless/#in-memory` (cit. a p. 23).

[12] *What is Apache Kafka?* URL: `https://www.tibco.com/reference-center/what-is-apache-kafka` (cit. a p. 25).

[13] *JGroups - A Toolkit for Reliable Messaging*. URL: `http://www.jgroups.org/overview.html` (cit. a p. 27).

[14] *Java Modelling Tools*. URL: `http://jmt.sourceforge.net` (cit. a p. 43).

[15] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi e Mani Subramanian. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns practices, 2013.

[16] *AxonIQ*. URL: `https://www.axoniq.io/`.

[17] Greg Young. «CQRS Documents». In: (nov. 2010). URL: `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`.

[18] Greg Woods. *Why would I need a specialized Event Store?* URL: `https://www.axoniq.io/blog/why-would-i-need-a-specialized-event-store`.

[19] *Domain Driven Design*. URL: `https://martinfowler.com/bliki/DomainDrivenDesign.html`.

[20] Seth Gilbert e Nancy Lynch. «Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services». In: (2002). URL: `https://dl.acm.org/doi/10.1145/564585.564601`.