Lagom Framework: Analisi del framework e implementazione di un progetto di esempio

Marco Baldassarri
marco.baldassarri2@studio.unibo.it

13 Agosto 2020

La grande crescita della complessità nello sviluppo dei prodotti software di qualsiasi natura e fine ha portato ad abbandonare la filosofia "monolite" che prevedeva l'uso di una serie di Client comunicanti con un unico Server in grado di gestire qualsiasi richiesta. Un approccio simile generava non pochi problemi di gestione delle risorse, ma soprattutto di complessità elevate nello sviluppo del software lato back-end per via della cattiva organizzazione del codice e la difficoltà di gestione di progetti troppo grandi, col rischio prevedibile di generare debito tecnico e di diminuirne la qualità.

Nella maggior parte dei contesti odierni si preferisce uno stile architetturale a Microservizi, in grado di sviluppare e distribuire una singola applicazione su una serie di servizi auspicabilmente di piccole dimensioni, ben distinti in quanto ruolo da svolgere ma al contempo in grado di comunicare, quindi distribuiti ma uniti. Tranne casi particolari, la modalità di comunicazione è la stessa dell'approccio Client-Server classico, dove non si faceva altro che esporre all'esterno una serie di API (disponibili tramite protocollo HTTP) la cui implementazione interna è nascosta al consumatore.

Nel mercato sono disponibili numerosi Framework di supporto alla creazione dei Microservizi, tutti aventi la pretesa di semplificare il più possibile la vita allo sviluppatore. Lagom è uno di questi. Il termine "Lagom" significa in Svedese "Just the right amount". Già dal nome si può intuire che i creatori di Lagom hanno voluto porre l'accento sulla definizione netta dei confini che ogni Microservizio deve avere. Ogni Service dovrebbe essere di piccole dimensioni e svolgere poche e semplici attività. La difficoltà risiede tuttavia nel capire come delineare la natura di ogni servizio, cosa far fare all'uno e cosa all'altro, e quindi creare un'architettura ben strutturata. Una buona ingegnerizzazione dei Microservizi porta a raggiungere più facilmente gli obiettivi di scalabilità e resilienza, ma anche di più facile distribuzione e gestione. Questo è il goal di Lagom.

Goal/Requirements : L'idea è quella di analizzare in maniera esaustiva il Framework Lagom ponendo l'enfasi anche sugli aspetti generali e teorici che vi sono dietro i vari temi ai quali la tecnologia afferisce, per poi terminare con un piccolo progetto di esempio dove vengono testate le varie funzionalità. Gli obiettivi preposti per questo progetto sono quindi i seguenti:

- Studio dell'architettura di Lagom e delle principali feature che il Framework mette a disposizione
- Comprensione delle tecnologie ausiliarie messe a disposizione da Lagom e fornire motivazioni delle scelte tecnologiche (Apache Kafka, Akka, Cassandra DB, Play Framework ed altri eventuali)
- Richiami teorici delle architetture residenti dietro l'infrastruttura Lagom, in particolare si vuole porre l'enfasi su:
  - Reactive programming
  - Microservices architecture
  - EventSourcing e CQRS
  - Actors (Akka)
  - Altri eventuali aspetti scoperti durante lo studio
- Confronto con altre tecnologie simili (e.g. Spring Framework)
- Valutazione dei limiti e dei punti di forza del Framework, cercando di capire in generale se e quando scegliere di usarlo per lo sviluppo di un prodotto software
- Implementare e testare le feature chiave di Lagom attraverso un piccolo progetto di esempio (una chat distribuita).

Work Plan :

- Studio della documentazione ufficiale e di altro materiale online che consentano la comprensione del Framework e dei principali casi d'uso
- Ricerche di articoli sugli aspetti teorici alla base del Framework
- Progettazione di un'app di chat distribuita facendo prima il design dei Microservizi in gioco e come essi interagiscono tra di loro
- Realizzazione dell'app che faccia uso delle principali feature di Lagom
- Creazione di Test per mettere sotto stress la tecnologia, usando un approccio TDD ove possibile

Contents

Terminologia

Framework: traducibile letteralmente come "cornice" o "struttura", un framework è un insieme di strumenti software atti a facilitare la progettazione e sviluppo di un applicativo software da parte di un programmatore. Può essere visto anche come un meta-linguaggio (ovvero un linguaggio derivante da un linguaggio esistente) che aiuta lo sviluppatore alla creazione della soluzione in maniera più semplice e veloce. Tale strumento non è da confondere con il concetto di Libreria: collezione di funzioni che svolgono verticalmente un singolo compito e collegate ad un programma software preesistente atte a favorire il riuso del codice come supporto allo sviluppo.
Publish-Subscribe: si tratta di un pattern basato sullo scambio di messaggi e viene principalmente utilizzato nei contesti IoT per interagire con i vari device ma anche per far comunicare più microservizi in un contesto back-end. Nel pattern publisher-subscriber, un messaggio pubblicato su un certo topic viene direttamente ricevuto da tutti i sottoscrittori in ascolto. Un topic è un modo per specificare dove pubblicare il messaggio, può essere visto quindi come un canale su cui vengono trasmessi i messaggi. E' rappresentato da un insieme di stringhe case sensitive aventi una precisa semantica e separate da forward-slash.

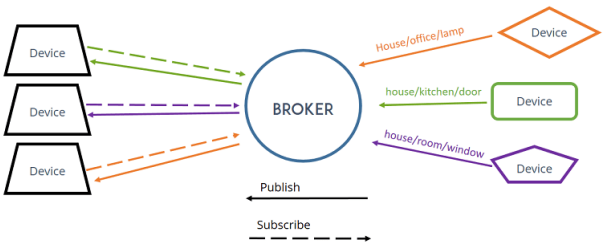

Figure 1: Example of Publish-Subscribe pattern in IoT context

Il messaggio non viene quindi spedito direttamente all'ascoltatore ma ad un "gestore" chiamato *Broker* o *Dispatcher*, il quale si occupa di ricevere tutti i messaggi, filtrarli e incanalari all'ascoltatore giusto. Il publisher non ha bisogno di conoscere l'uso che verrà fatto dell'informazione che spedisce così come il subscriber non ha bisogno di conoscere chi è il mittente del messaggio.

Nell'ambito della domotica e dell'Internet of Things l'uso di questo pattern semplifica la comunicazione tra sensori e favorisce una comunicazione leggera. Un protocollo famoso facente uso di questo pattern è MQTT (Message Queuing Telemetry Transport).
SBT e Maven: i build automation systems sono un insieme di script atti in primo luogo a processare la compilazione di un codice sorgente in file binari. Oltre che come supporto alla compilazione e alla creazione dell'artefatto, è uno strumento che si occupa di mantenere integre le versioni di tutte le librerie utilizzate, di lanciare test (sia in ambienti locali che virtuali creati al momento) e si presta bene quando combinato con la Continuous Integration. Ogni build infatti può essere automatizzato, e quindi messo in un Container o in una VM a seconda di come si è scelto di strutturare il flow che porta verso il delivery della soluzione. Usare tool di questo tipo porta a dei benefici non

indifferenti, come salvare tempo e denaro, avere un storico delle build e delle release, accelerazione dei processi e automatizzazione di task ridondanti.
Lagom utilizza SBT o Maven, a libera scelta.

Sbt è un tool open-source ed è lo strumento per eccellenza più utilizzato nella community Scala nonché dai creatori di Lagom e Play Framework. Ha pieno supporto per progetti scritti in Scala o Java, permette incremental testing, compilazione, e deployment.

Maven è un tool nato nel lontano 2004 sviluppato da Apache Software Foundation, è quindi maturo e anche se principalmente utilizzato per progetti Java, può essere utilizzato su altri linguaggi come ad esempio C#, Scala. Come linguaggio di scripting utilizza XML, a differenza di sbt, che utilizza un linguaggio molto simile a Scala.

Data store: In questo report per data store si intende un qualsiasi supporto harware/-software dove è possibile salvare dati in maniera persistente. A seconda dei contesti può essere una base dati relazionale o documentale, un file JSON, un file di testo e così via.

Domain Driven Design: introdotta per la prima volta nel 2004 da Eric Evans, DDD può essere definita come una buona pratica per ingegnerizzare un progetto software in accordo con i bisogni reali del dominio in costante evoluzione. Il focus del design è quindi sui domini e sulle entità di dominio.

DDD viene utilizzato nella risoluzione di problemi complessi, perché più grande è il progetto, più si sente la necessità di suddividere l'app in aree più piccole secondo le regole di business, noti come domini.

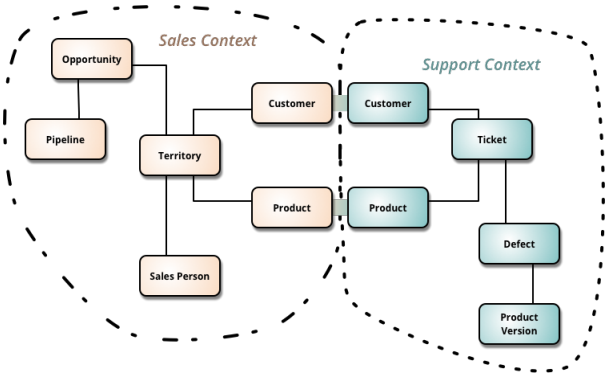

Figure 2: Example of DDD bounded contexts for an online store

Ogni sottodominio di un dominio più grande è noto sotto il nome di *bounded context*, ognuno avente un suo modello e delle sue entità. Una entità è un oggetto univoco all'interno del dominio. La sua unicità è ottenibile attraverso delle proprietà atte ad identificarla. Ad esempio, due utenti con lo stesso nome saranno distinti per mezzo degli indirizzi email o id o la combinazione di entrambi.

L'unico modo che hanno due bounded context di comunicare tra di loro o con il resto del mondo (senza creare inconsistenza) è attraverso un *aggregate root*, un oggetto che coordina gli altri oggetti del dominio. Questo tipo di design è utilizzato nel distribuito ed è alla base dell'architettura a Microservizi.

1 Lagom

1.1 Caratteristiche principali

Essendo un framework e non una semplice libreria, Lagom fornisce un supporto completo alla stesura dei microservizi fino ad arrivare al deployment. Un framework come Lagom permette di focalizzarsi sulla produttività evitando tutta la complessità che uno sviluppatore si trova a dover gestire nell'ingegnerizzazione di un sistema squisitamente distribuito, come ad esempio il requisito essenziale che tutti i nodi della piattaforma siano sincronizzati e non incorrano in problemi di inconsistenza dei dati e disponibilità.

Lagom permette la creazione di un template di base attraverso "sbt" oppure "maven". Dopo la creazione del template ci si accorge subito che tramite un unico file di configurazione di immediata comprensione è possibile definire i microservizi di cui si ha bisogno, nonché di metterli in esecuzione (soluzione molto comoda per un team di piccole dimensioni). In automatico appena dopo aver effettuato le modifiche, il build automation tool scelto si occupa di ricreare per noi tutto il template. Ogni microservizio è chiaramente distinto in due moduli principali all'interno dell'IDE: uno dove si può definire l'interfaccia e quindi i metodi che il service dovrà esporre, e uno dedicato all'implementazione di ogni metodo. Per progetti semplici gestiti da team di piccole dimensioni conviene un unico file per il building istantaneo di tutti i microservizi creati. Dalla documentazione si parla anche della possibilità di suddividere il building in file separati, così da proseguire autonomamente nello sviluppo di ogni singolo servizio integrandolo poi con gli altri durante la fase di release. Questa soluzione è consigliata per team e progetti di dimensioni più grandi e ovviamente aggiunge complessità. Fin da subito si notano quindi due aspetti fondamentali che Lagom ha messo a disposizione dello sviluppatore:

- Focus sul lavoro e non sulla configurazione per iniziare il lavoro.
  Dalla documentazione ufficiale spicca questa frase: "Lagom allows you to concentrate on satisfying your business needs."

- Separation of Concerns: lo scaffolding dell'applicazione di esempio (che vuole fungere anche da scheletro per la maggior parte dei progetti Lagom) permette una separazione netta tra interfacce ed implementazione di ogni servizio, nonché tra servizi, così da aumentare la qualità del codice e migliorarne la leggibilità.

Di seguito viene elencato un assaggio delle altre caratteristiche fondamentali che il framework mette a disposizione:

- Microservice Responsibility: La filosofia su cui si basa Lagom punta sul concetto di isolamento come prerequisito essenziale di maggiore resilienza ed elasticità. Ogni service viene visto come entità autonoma e responsabile del proprio stato. Due diversi microservizi non possono condividere il proprio stato: se un microservice cambia il proprio stato l'altro non ne è affetto. Citando Jonas Bonér, "What is needed is that each Microservice take sole responsibility for their own state and the persistence thereof." [1] Dichiarare un microservizio in Lagom è molto semplice ed intuitivo, è sufficiente dichiarare un `ServiceDescriptor`, il quale oltre che definire

in che modo il servizio sarà invocato e implementato, specifica con quale protocollo dovrà avvenire la comunicazione sottostante.

- Distributed Persistent Patterns: Lagom mette a disposizione più possibilità per rendere persistenti i dati, ma scoraggia l'utilizzo di un unico database centralizzato a favore della completa divisione delle responsabilità da parte di ogni microservizio; ognuno dovrebbe incapsulare il proprio stato e comportamento e di conseguenza un proprio modo di rendere persistenti i dati.
  Il framework incoraggia quindi l'utilizzo di un'architettura basata su Event Source e CQRS, fornendone una implementazione utilizzabile dallo sviluppatore chiamata `Persistent Entity`.

- Message Broker: per lo scambio e la condivisione dei dati tra microservizi viene messa a disposizione una libraria basata sul modello publish-subscribe, che permette la comunicazione attraverso canali di comunicazione chiamati topic. Lagom, inoltre, ha reso ogni topic *strongly typed*, in maniera che sia il produttore che il sottoscrittore del canale sanno in anticipo quale sarà il tipo di dato che verrà scambiato. Il topic viene dichiarato all'interno del `service descriptor` sopra citato.

- Polyglotism: Uno dei vantaggi di usare un'architettura a microservizi è il fatto che ognuno può avere una sua tecnologia e può essere implementato con un suo linguaggio. Lagom non pone restrizioni su questo, è perfettamente in grado di comunicare con altri microservices e viceversa. L'unico prerequisito è che per la comunicazione si utilizzi uno standard HTTP per le chiamate sincrone e un'architettura basata su WebSocket per lo scambio di messaggi asincrono e di streaming. Il parsing dei messaggi viene serializzato tramite JSON di default, fornendo una libreria apposita integrata, ma nulla vieta di impostare un differente linguaggio di serializzazione, come ad esempio XML.

2 Background Tecnologico e Metodologico

Lagom è stato costruito su tecnologie pre-esistenti e implementa dei pattern spesso usati nel distribuito. Di seguito si vogliono analizzare in dettagli tali aspetti al fine di comprendere il framework in maniera più approfondita.

2.1 Reactive Manifesto

Quando si parla di Reactive non ci si riferisce ad una tecnologia specifica, ma bensì di una qualsiasi tecnologia che abbia intrinsecamente adottato i principi del Reactive Manifesto.

Lagom vanta della fama di aver costruito internamente dei Microservizi di matrice *Reactive*. I reactive services sono di loro natura responsive (rispondono alle richieste molto velocemente), resilienti (tolleranti agli errori), elastici (facilmente scalabi sotto determinate condizioni) e orientate ai messaggi, quindi aventi un pattern di comunicazione asincrona tra microservizi.

Negli ultimi anni le tecnologie Reactive hanno preso piede soprattutto nel distribuito: le troviamo nelle più moderne librerie Core di Java 9+, come ad esempio la Reactive Streams API, ma anche progetti come come gli Akka Streams o RxJava. [15]

2.2 Microservices

2.2.1 Definizione

Non esiste una definizione ufficiale di questo stile architetturale, tuttavia si può riassumere come un modo di sviluppare un'applicazione dividendola in una suite di piccoli servizi, ognuno avente i suoi processi, il suo stato e il suo comportamento e perfettamente in grado di comunicare attraverso meccanismi leggeri, consumando una risorsa API esposta dal servizio attraverso HTTP, o sfruttando altre metodologie. In questo modo, ogni servizio fa una sola cosa, e la sa fare bene.

Come già citato, fino a qualche anno fa l'architettura backend più utilizzata era quella a monolite. Con l'avvento del cloud si è notato che ogni cambiamento anche solo ad una piccola parte del codice comportava dover ricreare una build e un deploy dell'intera applicazione, creando non poca frustrazione agli sviluppatori.

La risoluzione a questo problema sta nel dividerlo in parti più piccole, prendendo spunto dalla filosofia DDD (Domain Driven Design). Tale approccio viene adottato per sistemi complessi e suggerisce di dividere un modello troppo allargato e di difficile gestione in piccoli domini ben confinati secondo una certa logica, spesso rimandante alle *business capabilities*, cioè una divisione per contesti organizzativi aziendali (i.e. stock management, finance, payroll, orders etc.).

Il pattern principale di DDD è appunto *Bounded Context*, il quale focus è di delimitare ogni contesto, definendo per ognuno un suo specifico utilizzo e un modo di comunicare con gli altri contesti. Lagom punta molto su questo concetto, perché una buona definizione dei confini significa buona progettazione, e quindi minore debito tecnico nelle fasi di sviluppo successive.

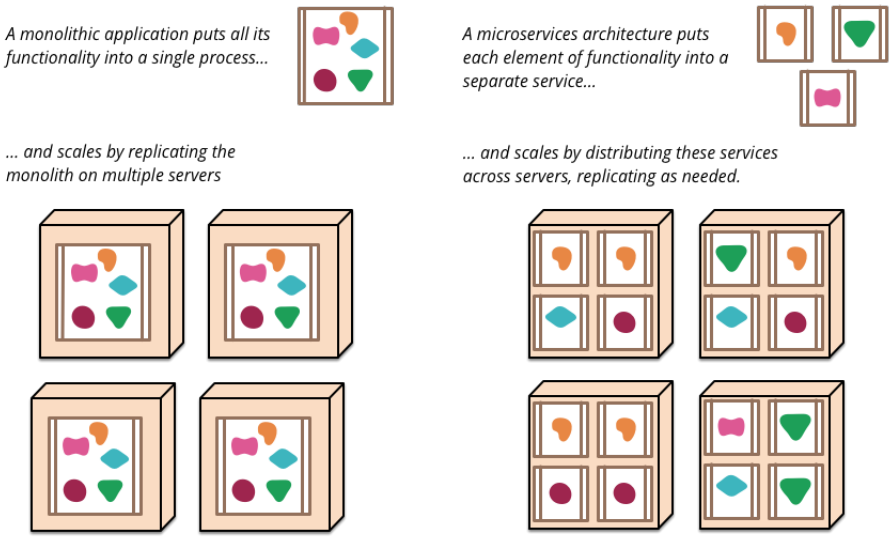

Figure 3: From monolith to Microservices [3]

2.2.2 Caratteristiche essenziali

Il principio chiave dell'architettura a microservizi risiede nella sua struttura squisitamente modulare delimitata da confini ben definiti: *Divide and Conquer*, ovvero la decomposizione del sistema in parti piccole, isolate e comunicanti attraverso protocolli specifici.

L'*isolamento* è una delle caratteristiche che un microservizio di qualità dovrebbe avere, perché prerequito di resilienza ed elasticità. L'isolamento tra servizi rende più naturale il *Continuous Delivery*, poiché permette di effettuare il deploy in produzione del singolo service e affidarsi ad un processo di sviluppo incrementale.

Per mezzo dell'isolamento di ogni servizio è anche possibile ottenere una migliore scalablità, nonché un monitoring, un debugging e un testing molto semplificati che sarebbero invece ardui nel caso di un unico monolite. Ad esempio, scovare un bug risulta più semplice, perché sarà sicuramente contenuto in un unico piccolo contesto dove non è necessario risalire a cascata attraverso i vari servizi per trovare il punto di rottura.

Jonas Bonér, co-fondatore e CTO di Lightbend, azienda che ha sviluppato Akka, Play Framework e Lagom, afferma che l'isolamento è il prerequisito per l'*autonomia*.

Un servizio autonomo è responsabile soltanto del proprio comportamento e ed è in grado di essere collaborativo comunicando con gli altri per mezzo di interfacce esposte verso il mondo esterno (API) e consumabili attraverso la Rete. Essere autonomi significa anche lavorare con promesse che il proprio comportamento porterà ad un preciso risultato. In questo modo sappiamo che ogni servizio è responsabile di uno specifico compito e non ha quindi bisogno di coordinamento e comunicazione con gli altri per svolgere le proprie azioni.

Altra caratteristica da sottolineare è il *poliglottismo*: a seconda delle esigenze, con

i microservizi è possibile mischiare diversi linguaggi, tecnologie, framework e database senza preoccuparsi delle dipendenze.

Infine, un altro dei punti di forza di questa architettura è che il *deploy* di un microservice è fattibile in un ambiente di produzione (cloud, server di qualsiasi tipo) senza particolari dipendenze dagli altri microservizi; si parla di *Indipendent Deployment* e questo evita probabilità di fallimento poiché il deploy di un sistema piccolo avente poche dipendenze è sicuramente meno "doloroso" di un sistema più complesso. Come già citato, tale processo viene spesso automatizzato attraverso un flow di CI/CD, che tra le altre cose si propone anche di testare adeguatamente ogni servizio prima della messa in opera. Si consiglia sempre di dispiegare ogni singolo microservizio in Cluster in modo che possa scalare aggiungendo risorse o togliendo risorse a seconda delle necessità, per renderlo più resiliente a fronte di maggiori richieste da parte degli utilizzatori o errori software che si presentano in fase di running. Così come i microservizi, anche i message broker e i data store, una volta messi in produzione, vanno resi ridondanti per evitare colli di bottiglia e single point of failure.

2.2.3 Nella vita reale

Nei prodotti reali non di rado si trovano dei servizi che fungono da sistemi di gestione centralizzata, degli orchestratori di tutti gli altri microservizi, sia per quanto riguarda le comunicazioni che per il salvataggio di dati, quindi viene adottato un approccio ibrido che si pone tra la mera architettura a microservizi e un sistema monolitico tradizionale. Ciò non toglie che i microservizi potrebbero essere visti anche come un insieme di Agenti indipendenti l'uno dall'altro (con un proprio ciclo di vita, un proprio stato e un proprio comportamento) i quali si limitano ad esporre al resto del mondo soltanto interfacce che agli altri potrebbe interessare al fine di scambiare dati per ottenere informazioni.

Nella pratica, per frammentare un'applicazione legacy monolite occorre porsi le seguenti domande importanti:

1. *Questo servizio svolge una sola funzione?*
   Se nello spacchettare un'applicazione di social network di tipo monolite abbiamo identificato un servizio che svolge la funzione di gestione degli amici e delle relazioni tra i vari amici, magari si potrebbe pensare di dividere ulteriormente in servizi più piccoli, uno che detiene il database di tutti gli amici, uno contenente un grafo raffigurante le connessioni tra i vari amici.

2. *Tale servizio è autonomo?*
   Come già detto, un servizio è responsabile di un suo comportamento e non dovrebbe dipende da altri. Esempio: abbiamo un servizio dedicato agli ordini di un articolo di magazzino e un servizio che si occupa della gestione dei pagamenti. Dopo che l'utente conferma l'ordine, il servizio degli ordini non si dovrebbe preoccupare se il pagamento è andato a buon fine, ma soltanto di inviare la richiesta di pagamento all'altro servizio. Magari, in maniera asincrona, prima o poi, riceverà una notifica di pagamento andato a buon fine. Questo è un servizio autonomo, la sua computazione continua a prescindere.

3. *Tale servizio detiene dei dati propri?*
   Ogni microservizio dovrebbe essere l'unico ad accedere al proprio database, sia in lettura che in scrittura, se così non fosse occorre riprendere il design dell'applicazione. Tuttavia va detto che nei sistemi reali non sempre ciò è possibile, ma per fortuna Lagom ha adottato una soluzione molto efficiente alla gestione della persistenza.

Il prezzo da pagare per poter utilizzare questa rivoluzionaria architettura può essere riassunto in una parola: *distribuito*. I microservizi non risiedono su una stessa macchina fisica ma bensì su nodi ben distinti della Rete, quindi non vi è condivisione di memoria. Le comunicazioni tra i vari servizi non sono semplici da sviluppare e vanno gestite adeguatamente, altrimenti possono incorrere a perdita dei dati, problemi di consistenza e Eventual Consistency (di cui si parlerà in seguito). Inoltre non dovrebbe mai mancare un buon team di DevOps in grado di gestire un numero considerevole di servizi su cui viene fatto il deploy regolarmente.

2.3 Communication in Lagom

In Lagom le comunicazioni tra microservizi possono essere implementate in varie modalità, ma è sempre consigliato utilizzare meccanismi asincroni per una maggiore efficienza e che siano orientati a mantenere isolamento ed autonomia tra di essi.

I due meccanismi principali sono le chiamate API e lo scambio di messaggi tramite broker con un approccio publish-subscribe. Di seguito verranno descritte più in dettaglio questi due soluzioni per effettuare la comunicazione.

2.3.1 Service Descriptors

In questo caso un servizio effettua una richiesta dei dati di cui ha bisogno all'altro servizio che li detiene, quindi il richiedente è un'entità attiva che si deve occupare di effettuare la chiamata e deve sapere chi contattare per poter ricevere i dati. Le chiamate al servizio vengono effettuate quindi in maniera sincrona e la comunicazione avviene per mezzo di interfacce (API) esposte agli altri microservizi ed accedibili per mezzo dei protocolli HTTP (ReST) e WebSocket. Questa soluzione è molto utilizzata anche per la comunicazione tra servizi e terze parti, ovvero client di qualsiasi tipo (front-end web, mobile app, device IoT)

Tali interfacce vanno adeguatamente definite, o per meglio dire, occorre descrivere in che modo vanno chiamate e come dovranno accedere al protocollo di trasporto sottostante.

```
2
import com.lightbend.lagom.javadsl.api.*;
4 import static com.lightbend.lagom.javadsl.api.Service.*;

6 public interface HelloService extends Service {
  ServiceCall<String, String> sayHello();
8
  default Descriptor descriptor() {
```

```
10      return named("hello").withCalls(call(this::sayHello));
    }
12  }
```

Listing 1: Service Descriptor simple example [11]

Implementare un Service Descriptor in Java è molto semplice: è sufficiente creare una `Interface` che estenda `Service` ed implementare il metodo di default `descriptor` che permette di dare un nome al servizio (una stringa univoca passata nel metodo `named`), definire quali funzionalità (comportamento) il servizio dovrà avere e in che modo verranno mappate sul transport layer.

Ogni interfaccia verso l'esterno definita del descriptor può essere identificata per mezzo di più tipologie di identificatori di chiamata (Call Identifiers). Il più semplice è quello del nome, una semplice stringa di testo che va passata al metodo `sayHello` e che nel caso sia implementato usando ReST sarà mappata con un path avente quel nome.

E' possibile identificare un metodo anche attraverso un URI passato al metodo `pathCall`, il quale permette anche una sintassi speciale che utilizza i due punti ":" per poter accettare stringhe dinamiche (ad esempio con il path `/order/:id` chi utilizza l'interfaccia può passare un id specifico alfanumerico).

Infine Lagom mette a disposizione l'identificazione di un nuovo comportamento tramite ReST call, con un metodo molto simile al precedente, ma dove è possibile anche specificare il codice HTTP. Quest'ultimo viene usato soprattutto quando si usa HTTP ReST per la comunicazione e si vuole dare una semantica aggiuntiva all'API definita (oltre al path, che già di per sé contiene stringhe significative). Di seguito sono riportati gli esempi delle tre modalità appena descritte.

```
//named based identifier
2   default Descriptor descriptor() {
      return named("hello").withCalls(namedCall("hello", this::sayHello));
4   }


6
    [...]
8


10  //path based identifiers
    ServiceCall<NotUsed, Order> getOrder(long id);
12
    default Descriptor descriptor() {
14    return named("orders").withCalls(pathCall("/order/:id", this::getOrder));
    }
16


18  [...]


20
    //REST based identifiers
22  ServiceCall<Item, NotUsed> addItem(long orderId);
    ServiceCall<NotUsed, Item> getItem(long orderId, String itemId);
24  ServiceCall<NotUsed, NotUsed> deleteItem(long orderId, String itemId);
```

```
26 default Descriptor descriptor() {
   return named("orders")
28       .withCalls(
             restCall(Method.POST, "/order/:orderId/item", this::addItem),
30           restCall(Method.GET, "/order/:orderId/item/:itemId", this::
     getItem),
             restCall(Method.DELETE, "/order/:orderId/item/:itemId", this::
     deleteItem));
```

Listing 2: Call Identifiers types

Una volta definiti i metodi esposti, in un'altra classe si andranno ad implementare così da poter gestire il comportamento che deve avere quel microservizio a seguito di una chiamata dall'esterno.

2.3.2 Message Broker

Un altro modo molto utilizzato per far comunicare i microservizi è quello di adottare il pattern publish-subscribe, il quale permette a questi ultimi di condividere dati attraverso dei topic, pubblicare dei "messaggi" ad un broker che si occuperà di demandarli al microservizio in ascolto. In questo caso quindi se un servizio ha bisogno dei dati di un altro servizio non li richiede direttamente ma si avvale di un intermediario che salva i dati per un tempo predefinito: questa frammentazione ulteriore dei componenti in gioco fa sì che i momenti in cui si pubblicano e si consumano i dati possano essere diversi, predisponendosi così ad una comunicazione asincrona. Quando la comunicazione avviene in maniera sincrona vuol dire che tutte le entità in gioco devono essere attive nello stesso momento. Ciò porta a problemi di inconsistenza, poiché basta che solo una delle parti fallisca per far sì che la comunicazione venga interrotta e il dato completamente perso.

Il modello di "Reactive Microservice" su cui si basa Lagom suggerisce di adottare il più possibile una comunicazione asincrona, per questo mette a disposizione anche delle librerie atte a gestire questo tipo di approccio. Le Message Broker API sono state implementate in modo da garantire che tutti i messaggi arrivino a destinazione e che ai nuovi sottoscrittori appena entrati vengano recapitati tutti gli eventi passati oltre che quelli presenti e futuri. Queste API forniscono un'implementazione del modello publish-subscribe dove è possibile condividere dati attraverso un topic e facendo uso di un broker per favorire la comunicazione asincrona.

Un topic è un canale che permette ai vari servizi di pubblicare o recapitare dati. Lagom forza l'associazione del topic ad un tipo, in questa maniera sia il produttore che il sottoscrittore del messaggio conoscono in anticipo (prima di ricevere il dato) la natura del dato stesso. Il topic viene dichiarato all'interno del metodo `descriptor` sopra citato e i dati passati attraverso di esso saranno tutti serializzati in JSON come modalità predefinita, ma è possibile utilizzare un parser differente.

Il framework mette a disposizione anche un broker predefinito chiamato Kafka. Kafka è un broker molto utilizzato in ambito industriale soprattutto da aziende aventi prodotti dove è di vitale importanza la gestione in tempo reale dei flussi di informazioni e quindi di grandi quantità di dati (si citano Twitter, Netflix, LinkedIn e altri).

Tale componente può essere descritto come un accumulatore di eventi distribuito che salva in una coda i nuovi record in arrivo. La scelta di Lightbend del broker ottimale è ricaduta su Kafka (anziché altre tecnologie come ad esempio RabbitMQ) perché oltre che fungere da smistatore di messaggi ha una politica di ritenzione, ovvero conserva i messaggi in memoria per un certo periodo di tempo in caso di necessità (fault tolerance) aggiudicandosi anche il titolo di *database* oltre che semplice broker. I topic hanno una semantica, hanno il potere di far suddividere in categorie i messaggi scambiati. Kafka esegue un'ulteriore suddivisione, ovvero distribuisce i messaggi di un certo topic lungo un numero specificato di *partizioni*, viste come una sequenza immutabile di record. Questo favorisce scalabilità, in quanto più consumer possono accedere a partizioni diverse dello stesso topic. A causa di questo partizionamento, i messaggi potrebbero essere processati senza un ordinamento desiderato. Se stiamo sviluppando un'applicazione dove l'ordinamento è importante, dobbiamo assicurarci che ogni messaggio pubblicato su un certo topic vada a finire nella partizione giusta. La libreria Message Broker API viene in aiuto permettendo di configurare una strategia di partizionamento che verrà incapsulata nel messaggio stesso prima che sia spedito. Kafka farà uso di questa chiave per aiutarsi a decidere su quale partizione spedire ogni messaggio. Per dichiarare un topic è necessario scomodare di nuovo il Service Descriptor, in particolare nel pattern builder interno al metodo `descriptor` verrà chiamato un `withTopics` che definirà l'insieme dei topic che il microservizio utilizzerà per pubblicare i propri dati. Di seguito un semplice esempio di dichiarazione, dove in aggiunta si può notare la strategia di partizionamento di Kafka.

```
return named("blogpostservice")
    .withTopics(
        topic("blogposts", this::blogPostEvents)
            .withProperty(KafkaProperties.partitionKeyStrategy(),
                BlogPostEvent::getPostId));
```

Listing 3: Topic declaration example with partition strategy

2.3.3 Clustering

In aggiunta a queste due soluzioni principali per far comunicare i microservizi, esistono dei meccanismi che Lagom suggerisce per la comunicazione "intra-service", ovvero tra nodi interni dello stesso servizio, genericamente chiamati *cluster*, e spesso creati per garantire scalabilità. In questo caso vi è la condivisione dello stesso codice e vengono gestiti insieme come un set, quindi non vi è un eccessivo bisogno di frammentazione dei componenti. Tali meccanismi interni ai componenti di Lagom includono *Akka Remoting* e i database (in particolare se di tipo Event Sourcing).

Di persistenza si parlerà largamente nei prossimi paragrafi ma in generale è possibile vedere i database (e quindi lo storage condiviso) come un modo per far comunicare i nodi di un microservizio. Se poi non sono relazionali ma di tipo Event Sourcing verrà garantita la comunicazione asincrona, con tutti i vantaggi che ne deriva.

Akka Remoting è un modulo di comunicazione presente nel Framework Akka che si avvale della connessione peer-to-peer tra i vari Attori. La comunicazione in questo caso

si presuppone sia simmetrica: se un sistema A se si connette con un sistema B allora anche il sistema B si dovrà poter connettere al sistema A. Non deve esistere un sistema che accetta connessioni senza fare altro, come non deve esistere un sistema che stabilisce connessioni senza fare altro. Questo risultato non è sempre raggiungibile per via di problemi relativi alla Rete, come nodi dove è presente NAT, Firewall, o in casi dove si usano i container di Docker non configurati per aprire le comunicazioni verso l'esterno. Questo setup è chiaramente molto diverso da un tipo di interazione client-server (come invece avviene per i Service Descriptors di cui sopra) poiché in questo caso i ruoli sono ben definiti: in questo caso si preferisce l'uso di HTTP anziché avvalersi del framework di Akka.

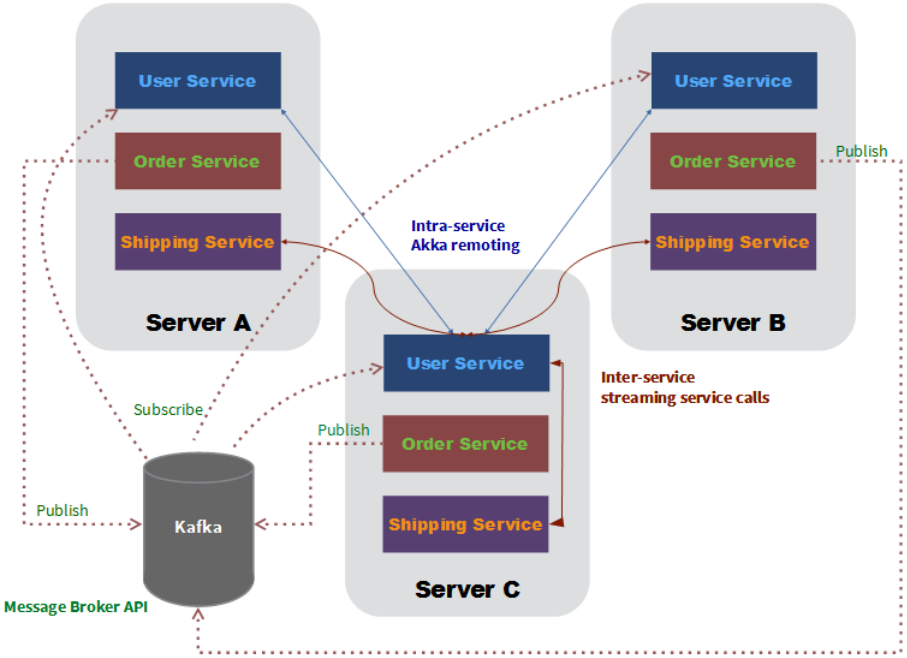

Figure 4: Communication between microservices in Lagom

Il diagramma illustra un esempio dell'utilizzo delle tecnologie citate per la comunicazione intra e inter service distribuita lungo tre server dedicati: un servizio Order pubblica dei dati su un topic Kafka (freccia tratteggiata), mentre il servizio User diventa un sottoscrittore e resta in ascolto dei messaggi. Trattandosi di un servizio con overhead di richieste elevato, User service è stato diviso in più istanze creando così un cluster. La comunicazione tra ogni istanza avviene tramite Akka Remoting (freccia blu), mentre Shipping service e User service comunicano attraverso service calls (freccia rossa).

2.4 Actors and Akka

Un modello ad attori è un modello concettuale utile nella computazione di sistemi concorrenti; definisce regole generali su come vari componenti in gioco dovrebbero interagire

tra di loro. Il linguaggio più famoso che implementa il modello ad Attori è senza dubbio *Erlang*. Un attore può essere definito quindi una unità minima di computazione, può ricevere messaggi e reagire ad essi svolgendo delle computazioni interne. L'idea è molto simile a quella dei linguaggi object-oriented: un oggetto riceve un messaggio (in questo caso una chiamata di metodo) e svolge una azione di un certo tipo come conseguenza della ricezione dello stesso. La differenza sostanziale è che gli Attori sono completamente isolati tra di loro e non condivideranno mai la memoria. Inoltre, un Attore detiene un proprio stato interno che nessun altro Attore esterno potrà mai cambiare. Gli Attori possono spedire messaggi in maniera del tutto asincrona ad altri Attori, i quali mantengono una Mailbox dove vengono salvati momentaneamente i messaggi in arrivo in attesa di essere processati. L'organizzazione del modello ad Attori è gerarchica: un attore può creare altri attori e assegnare loro dei sub-task da svolgere per permettere una corretta suddivisione dei compiti.

Akka è un framework open-source sviluppato da Lightbend (la stessa che ha sviluppato Lagom), con cui è possibile costruire programmi basati sul modello ad Attori.

Si tratta di una tecnologia ormai matura e largamente utilizzata a livello industriale per la costruzione di sistemi concorrenti e distribuiti basati sulla JVM. Non a caso la maggior parte dei componenti di Akka sono stati utilizzati per la costruzione di Lagom. Nell'utilizzare Lagom si ha il privilegio di lavorare molto ad alto livello per la costruzione di microservizi, quindi non si deve interagisce direttamente con gli Actors a basso livello, ma nello sviluppare si ha comunque la sensazione di avere un motore Akka sottostante. Nulla vieta, nello sviluppare usando Lagom, di interfacciarsi direttamente con un ActorSystem object (iniettato tramite Dependency Injection) per poter dialogare direttamente con Akka e gli Attori, ma questa possibilità è utilizzata soltanto in contesti molto particolari. [7]

2.5 Play Framework

La stessa azienda che ha prodotto Lagom è anche la fondatrice del più famoso Play Framework, principalmente destinato al supporto dello sviluppo di applicazioni Web con pattern MVC.

Anch'esso molto leggero e basato su Akka e Akka Streams per fornire un utilizzo minimo delle risorse in termini di memoria, thread e CPU. Scritto in Scala ma utilizzabile anche con Java, anche Play come tutti gli altri framework di rispetto permette un interfacciamento semplificato ai database relazionali e non relazionali a libera scelta dell'utente per mezzo di uno strato di ORM.

Lo sviluppatore Lagom non si trova ad interagire direttamente con le librerie di Play, quindi la sua presenza rimane del tutto trasparente a meno di particolari interfacciamenti dei microservizi di Lagom con prodotti software preesistenti basati su Play.

2.6 CQRS

> "At its heart is the notion that you can use a different model to update information than the model you use to read information" — Martin Fowler

[2]

> "The Command and Query Responsibility Segregation (CQRS) pattern separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security. The flexibility created by migrating to CQRS allows a system to better evolve over time and prevents update commands from causing merge conflicts at the domain level." [13]

Nella maggior parte degli applicativi tradizionali il modello più utilizzato per le operazioni di lettura/scrittura è quello CRUD (Create, Read, Update, Delete). Usare questo approccio presenta delle limitazioni che si notano soprattutto in contesti dove le applicazioni da sviluppare hanno una business logic non banale. I limiti più comuni sono per lo più concernenti conflitti nell'update dei dati, specie se in un contesto in cui più utenti concorrentemente ne fanno uso cercando di accedere alla stessa risorsa. Oltre a ciò non manca da sottolineare il fattore performance e quindi l'overhead a cui possono portare le operazioni di update direttamente mirate al data store.

Per facilitare la vita allo sviluppatore, quando la base dati viene interrogata i dati restituiti vengono modellati tramite ORM (Object Relational Mapping) per renderli fruibili come oggetti interpretabili dai più comuni linguaggi di programmazione, e quindi anche dai framework. Spesso anche questa operazione di mapping può diventare complicata e incrementare esponenzialmente la complessità della business logic dell'applicazione. L'approccio CRUD non si presta bene quindi per progetti di grandi dimensioni, anche per problemi di performance e scalabilità.

Il pattern CQRS si occupa quindi di indirizzare *read* e *write* in due modelli separati, usando dei comandi per fare l'update del dato e delle query per leggerlo. Questo concetto si evince già dall'acronimo di CQRS che sta per *"Command Query Responsibility Segregation"*, che letteralmente si può tradurre come "Suddivisione della responsabilità tra Comandi e Query".

Un'azione di un utente è rappresentata da un comando che viene dato all'applicazione. Associamo la parola *Command* a ciò che vogliamo fare, ad una azione compiuta dall'utente o da qualsiasi altra entità che porta ad un cambiamento di stato nel nostro sistema (es. add like to the tweet, accept payment, upload new picture..). Il comando porta a scatenare un numero di eventi che saranno poi resi persistenti.

Avere modelli separati per query (read) e update (write) semplifica il design e l'implementazione, inoltre le performance vengono migliorate in maniera considerevole. L'isolamento delle parti è quindi la chiave del pattern CQRS. Ad esempio, un approccio comune è quello di creare un database relazionale per le operazioni di update e un altro database separato (magari document based e denormalizzato) per creare delle viste efficienti e ottimizzate per le letture.

Tra i più importanti benefici di CQRS troviamo:

- **Indipendent Scaling**: i carichi di lavoro delle read e delle write scalano in maniera indipendente e riducono drasticamente la corsa critica per ottenere il *lock*.

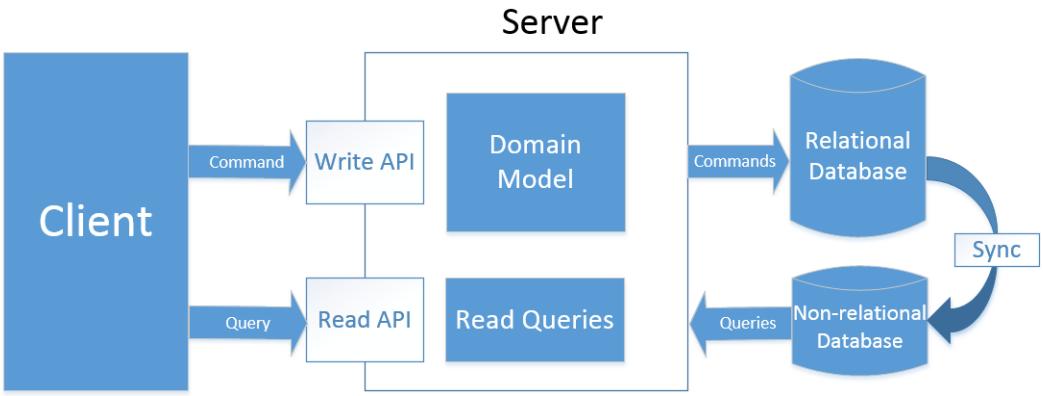

Figure 5: CQRS with separate storage

- **Data Schemas ottimizzati**: il modello che si occupa delle read può usare uno schema ottimizzato per le query, mentre il modello che si occupa delle write usa uno schema ottimizzato per effettuare gli aggiornamenti in maniera efficiente.
- **Security**: è più semplice verificare quali siano le entità di dominio che effettuano le scritture sui dati.
- **Separation of Concerns**: separare le scritture dalle letture può portare ad un'architettura dei modelli più semplice e flessibile. La maggior parte della business logic è scritta all'interno del write model, il read model può risultare relativamente semplice.
- **Query semplificate**: avere un database separato per le read permette evitare operazioni complesse quando si effettua la query grazie a tabelle ottimizzate per effettuare operazioni di read, quindi per le query. Come già accennato in precedenza, il database pensato per le letture è spesso denormalizzato, proprio allo scopo di evitare l'uso di `Join`, operazione considerata molto dispendiosa soprattutto se effettuata tra un numero elevato di tabelle. Si evitano quindi quei problemi tipicamente riscontrati in un approccio CRUD.

Come tutti i pattern architetturali anche questo presenta delle considerazioni da fare che potrebbero indurre a problemi sia nello sviluppo che in produzione. Ad esempio, è importare sottolineare che avendo diviso la controparte di *command/update* da quella di *lettura*, si introduce un bisogno di mantenere i due modelli consistenti tra di loro.
In CQRS, i due modelli di read e write devono essere mantenuti sincronizzati in qualche modo. Solitamente questo obiettivo è raggiunto facendo pushare al modello di write degli *eventi* caratterizzanti la modifica del proprio database. Ogni qualvolta il write data store subisce una modifica quindi, spedisce un messaggio di notifica al modello di read, il quale aggiorna il proprio database. Questa operazione deve avvenire, per quanto possibile, in una singola transazione, per evitare inconsistenze del dato. Purtroppo è

molto difficile a volte capire se ciò che un utente visualizza a video (dopo aver effettuato una lettura) è un dato corretto o meno. Usando una comunicazione dei vari eventi tipicamente a messaggi, essi possono andare persi, duplicati ecc, quindi l'applicazione deve poter gestire anche i message failures. Questo problema è noto nei sistemi distribuiti con il nome di "Eventual Consistency", ovvero il garantire *prima o poi* che tutti i nodi di un database, o di un pool di microservizi o altro, siano sincronizzati e i dati al loro interno replicati adeguatamente.[14] Il problema risiede per l'appunto nel tempo di aggiornamento tra i due modelli. In alcune applicazioni real time è di vitale importanza per l'utente che fa delle query in lettura avere un dato fresco a disposizione quanto prima. Con un approccio CQRS questo aspetto potrebbe essere critico.

Un altro elemento da citare è la complessità a cui porta l'adozione di questo pattern: l'idea di CQRS è pulita ed elegante ma è complessa da implementare: può richiedere un lavoro non indifferente allo sviluppatore.

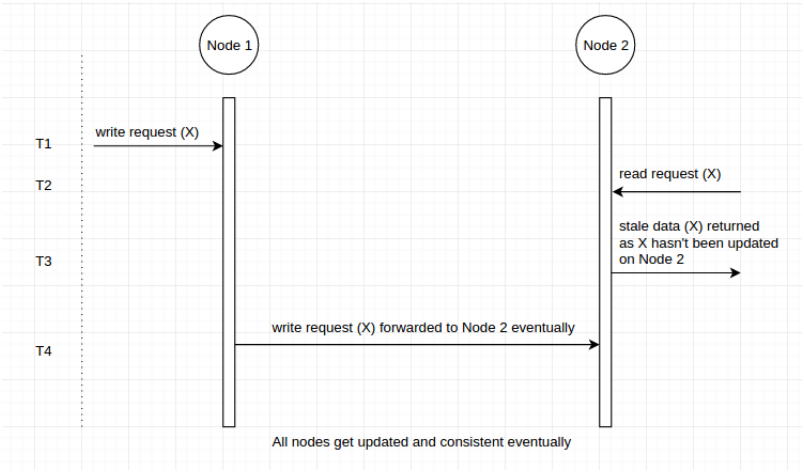

Figure 6: Eventual Consistency

2.7 Event Sourcing

> "Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes" — Martin Fowler

> "Event Sourcing is the practice of capturing all changes as domain events, which are immutable facts of things that have happened." — Lagom offical Documentation

Del pattern Event Sourcing se ne discuteva già negli anni '70 ma solo oggi sta prendendo piede grazie alla programmazione asincrona.

In generale, il pattern Event Sourcing definisce un approccio atto a gestire le operazioni sui dati per mezzo di una sequenza di eventi, ognuno dei quali è registrato in uno store di sola scrittura.

Come già accennato uno dei problemi di CQRS è che una volta separati i modelli di read e write nelle transazioni di un db, essi devono essere mantenuti in sync.

E' proprio qui che entra in gioco l'Event Sourcing: ogni evento rappresenta un cambiamento sul dato, o meglio, ogni operazione sul dato è guidata da una sequenza di eventi, descriventi le azioni compiute.

Si può immaginare gli *Events* come cose che accadono, azioni compiute immutabili (i.e. tweet added, user logged etc). Ogni evento rappresenta quindi un insieme di cambiamenti del dato. Ogni qualvolta un'entità cambia di stato, un nuovo evento viene appeso alla sequenza di eventi. A differenza dell'approccio più tradizionale, che si occupa di cambiare lo stato ad un entità e rendere il cambiamento persistente, Event Sourcing adotta una filosofia *event-centric* alla persistenza, dove è possibile quindi cambiare lo stato ad un dato e pubblicare l'evento relativo al cambiamento in maniera del tutto

atomica. In questo contesto il database viene visto quindi come un "append-only log of serialized events" [5]

Tramite Event Sourcing, gli stessi eventi possono essere utilizzati per notificare altri componenti come ad esempio il modello read (nel caso di unione con CQRS), il quale usa il log della sequenza di eventi per ricreare lo stato corrente e mantenersi sincronizzato con il write. La sequenza di eventi solitamente viene mantenuta in un'entità molto importante chiamata *event store*, la quale mantiene lo stato corrente del dato.

Esso funge anche da dispatcher di eventi comportandosi come un message broker, ovvero notifica tutti i consumatori di eventi che restano in ascolto e fornisce una API per permettere a tutti gli ascoltatori di sottoscriversi agli eventi.
Un uso tipico è quello delle interfacce utente, che per avere dati sempre freschi fanno uso del'event store. Gli eventi possono essere rappresentati come oggetti contenenti le azioni scatenate dagli utenti che hanno portato ad un cambiamento di stato del dato e che necessitano un refresh nella View per rappresentare visivamente l'azione rappresentata nell'evento.

Il pattern Event Sourcing può fornire i seguenti vantaggi:

- E' possibile effettuare un auditing accurato grazie ai log storici e quindi ricostruire lo stato del dato con grande accuratezza. Immaginando l'event store come uno storage di eventi continuamente appesi in maniera sequenziale, è possibile anche fare analisi sul comportamento e l'andamento dei dati, in modo da ottenere informazioni molto utili a livello di business (ad esempio il trend degli acquisti di un determinato prodotto ecc).

- Viene eliminata gran parte della complessità dei database relazionali, soprattutto per quanto riguarda gli update concorrenti (che potenzialmente possono creare conflitti) a favore di un improving delle performance e scalabilità. Questo è possibile grazie al fatto che non vi è la necessità di modificare direttamente l'oggetto nel data store, ma tutto viene gestito per mezzo di nuovi eventi descriventi la modifica. In altre parole, l'update nei sistemi tradizionali è un'operazione "distruttiva", mentre qui si tratta di un update "incrementale".
  Per fare un esempio, la variabile `int 50` dopo un update viene sostituita con `51` poi sostituita con `52` e così via, perdendo anche la storia degli stati precedenti.
  Con Event Sourcing, avendo come stato attuale una variabile `int 50`, l'update diventa un comando (command) che dice di aggiungere `+1`, al quale viene agganciato un altro comando che impone di aggiungere `+1`, avendo come stato attuale `52`.
  In questo contesto abbiamo uno storico degli stati precedenti di quella variabile.

- Abbiamo da una parte l'event store che si occupa di fornire eventi, e dall'altra uno o più task che portano a termine qualche operazione in risposta a tali eventi. Questa divisione dei lavori porta a maggiore flessibilità ed estendibilità, poiché diventa più semplice l'integrazione di altri servizi in ascolto degli stessi eventi.

- Miglioramento nelle performance delle scritture, grazie al fatto che ad ogni cambiamento nuovi eventi vengono appesi al data store formando uno stream, senza andare a cambiare direttamente il dato contenuto nell'evento.

- Miglioramento nelle fasi di testing e debugging dell'applicazione. Gli eventi e i comandi possono essere facilmente simulati e in produzione è più facile scovare bug facendo una semplice replica dei log in un ambiente controllato

Non ci si può astenere dal fare anche delle considerazioni, non viste come degli svantaggi, ma sicuramente concetti da non sottovalutare riguardo tale pattern:

- In un contesto distribuito, occorre prendere in considerazione il fatto che potrebbe esservi un delay in termini di tempo tra l'agente che aggiunge l'evento all'event store, la pubblicazione dell'evento, e la ricezione da parte del consumatore che deve gestirlo. Durante questo periodo, nuovi eventi con la stessa semantica (ad esempio quelli riguardanti gli ordini) potrebbero essere arrivati all'event store ma avere al loro interno un dato aggiornato, quindi uno stato diverso.

- Il dato stesso contenuto nell'evento non dovrebbe essere mai modificato, se non per mezzo di un nuovo evento che descrive il cambiamento di stato. Potrebbe essere necessario aggiungere dei timestamp e allegarli ad ogni evento, altrimenti si potrebbe creare inconsistenza.

- In un contesto multi threaded, più thread potrebbero salvare gli stessi eventi nell'event store. E' utile ribadire che la consistenza di quest'ultimo è di vitale importanza perché tutto funzioni correttamente, quindi occorre un'attenzione maggiore da parte dello sviluppatore.

- In architetture mature e molto utilizzate come ReSTful, sono stati eletti dei parser per lettura e l'estrazione dei dati (Json e XML). Nel contesto Event Sourcing non esiste un approccio standard per la lettura del dato e per estrapolarne l'informazione.

- Anche se Event Sourcing riduce la possibilità di mandare in conflitto l'update dei dati, ci potrebbe comunque essere inconsistenza negli eventi data dal fatto che due eventi capitano allo stesso momento. Ad esempio, mentre un evento che viene inviato all'event store descrive una riduzione della quantità in magazzino di un certo prodotto, nello stesso momento potrebbe arrivare un evento di un ordine da parte di un utente che supera il numero degli item effettivamente disponibili.

2.8 Event Sourcing e CQRS

In molti casi è utile far convivere entrambi i pattern descritti, usando CQRS per il separation of concern e Event Sourcing per permettere una comunicazione ottimizzata tra il modello di read e write.

Il modello write di un sistema basato su CQRS diventa lo store degli eventi e quindi la fonte ufficiale da dove vengono prese tutte le informazioni.
Il modello di read diventa un *presenter* del dato, quello incaricato a mostrarlo in maniera efficiente quando gli viene richiesto. Il modello di read ad esempio, in un contesto a Microservizi, potrebbe essere un microservice dislocato geograficamente in un altro nodo della Rete che si occupa soltanto di fornire i dati in lettura nel migliore dei modi in termini di performance.

Per fare un altro esempio, si pensi alla tabella `Orders` in un database relazionale, contenente un id, uno stato dell'ordine e un amount totale. Questa tabella, tramutata in un event store nell'ambito Event Sourcing, descrive tali dati come azioni contenenti i dati stessi, quindi in esso si troveranno eventi salvati con nominativi simili a `OrderCreated`, `OrderApproved`, `OrderShipped` e così via. Dato che adottiamo anche un pattern CQRS, tale event store potrebbe trovarsi incapsulato in un microservice dedicato alle scritture e continua a ricevere eventi. Ogni evento citato conterrà abbastanza dati da permettere all'ascoltatore di ricostruire successivamente lo stato dell'Ordine. L'ascoltatore potrebbe in prima battuta essere un altro microservizio dedicato al modello read che si occupa di sincronizzarsi con gli eventi arrivati nell'altro microservizio, denormalizzarli e renderli fruibili a chi poi dovrà fare le query per leggerli, come ad esempio un'interfaccia grafica.

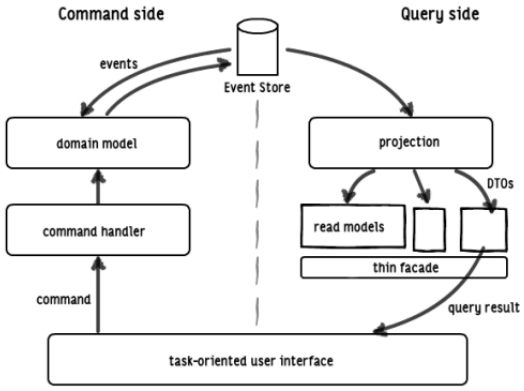

Figure 7: Interaction blocks describing CQRS and Event Sourcing togheter [12]

Ovviamente anche unendo i due pattern i pro e i contro di ognuno rimangono. Come già riportato in precedenza, i due modelli di CQRS devono essere sincronizzati, quindi è opportuno considerare il delay tra il modello di write, che detiene il database degli eventi e potrebbe essere implementato, ad esempio da un microservizio A, e la propagazione degli eventi di A sul modello di read, rappresentato ad esempio da un altro microservizio B. Tale microservice deve essere visto come una *copia read-only del write model*.
Nulla ci vieta di avere più repliche di read in ascolto degli eventi che il microservizio A spedisce, al fine di poter avere più performance nelle letture. Un ulteriore problema potrebbe risiedere nella quantità elevata di risorse necessarie a processare tutti gli eventi in arrivo per estrapolare il dato e renderlo disponibile per le read. Questo problema è ovviabile creando degli snapshot dello stato dei dati ad intervalli regolari.

2.9 Persistence in Lagom

2.9.1 PersistentEntity

In Lagom la classe astratta `PersistentEntity<Command,Event,State>` rappresenta un'entità avente un'idendificatore univoco (primary key) che viene utilizzato per accedere

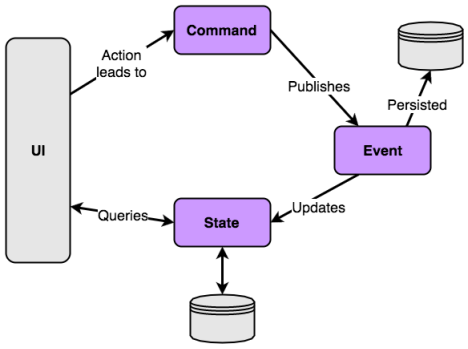

Figure 8: CQRS flow example

```
AGGREGATES
{
  food:   [ burger, salad ],
  drinks: [ beer, water ]
}

COMMANDS
{ addWater  }
{ addSalad  }
{ addBeer   }
{ addBurger }

EVENTS
{ waterAdded  }
{ saladAdded  }
{ beerAdded   }
{ burgerAdded }
```

Figure 9: Data example

allo stato corrente dell'evento generato. Ogni identificatore è univoco e rappresenta un'unica istanza dell'entità. Lagom si preoccupa di distribuire le istanze di ogni identità lungo il cluster di microservizi. Conoscendo l'identificatore è possibile spedire messaggi (o detta alla CQRS, impartire comandi) all'entità. Contiene tre membri generici chiamati `Command, Event e State`. I primi due sono gli stessi già descritti in precedenza, mentre lo State rappresenta la condizione di un'entità in una specifica istanza.

Allineandosi con il flow di CQRS descritto in Figura 8, tutte le interazioni procedono spedendo un command ad un'entità alla quale segue un update oppure un messaggio di risposta contenente il dato richiesto. Questo significa che anche la query per ottenere lo stato corrente è gestita spedendo un command. Nel caso di intenzione di effettuare una modifica allo stato, il command produrrà un event che sarà reso persistente (attraverso Cassandra o altre soluzioni di storage) e produrrà un cambio dello state.

In termini di DDD (Domain Driven Design) l'oggetto `PersistentEntity` è un "Aggregate Root". Un aggregate si può immaginare come un'unità di incapsulamento dello stato, una classe che detiene tutti i cambiamenti di stato. Concretamente un aggregate si occupa di gestire i commands (quindi risponde a query per uno specifico identifier, proprio come fa PersistentEntity in Lagom) e di applicare gli eventi in arrivo; ha un proprio modello descrivente lo stato corrente che cambia a seconda delle indicazioni del command. Un aggregate forma un albero o un grafo di relazioni. La root dell'albero è detta aggregate root. Ogni aggregate possiede quindi una root entity: si tratta dell'unico elemento presente nell'aggregate capace di dialogare con gli oggetti esterni, quindi col resto del mondo. Ogni oggetto esterno all'aggregate ha il root entity come unico riferimento. La Figura 9 mostra un semplice esempio dello stato di un aggregate derivato da comandi ed eventi.

2.9.2 Cassandra

Di default (cioè senza bisogno di ulteriori installazioni) il framework Lagom mette a disposizione Apache Cassandra per la persistenza, fornendo di fatto una API completamente asincrona per salvare i dati, favorendo quindi maggiore scalabilità e una minore

latenza agli utenti finali che utilizzano il prodotto software. Cassandra è un database management system di tipo NoSQL, è open-source, ed è stato costruito per gestire un numero molto grande di dati in maniera distribuita lungo una molteplicità di server, fornendo alta disponibilità e nessun point-of-failure. Per chi preferisce una soluzione sincrona Lagom dalle ultime versioni fornisce supporto a JDBC o JPA, dove però è opportuno gestire manualmente il pool di thread che si occupano di chiamare queste API bloccanti. Tramite `sbt` è sufficiente specificare che uno dei nostri microservizi fanno uso del database e opzionalmente fornire la porta per ai driver di interfacciarsi con il server Cassandra. Si consiglia di specificare manualmente una porta per evitare interferenze con altre istanze di Cassandra presenti nella propria macchina.

3 Progetto Lagom Chat

Per rendere più tangibili i concetti sopra descritti, ho deciso di implementare un progetto di esempio in Java che copra più concetti possibili messi a disposizione dal framework Lagom. Il goal principale è stato quindi quello di riuscire ad implementare l'Event Sourcing in Lagom, renderlo funzionante e testarlo per quanto possibile. Si è voluto focalizzare l'attenzione anche sulla comunicazione tra microservizi e sull'investigazione delle varie tipologie di interazione che il framework mette a disposizione. L'outcome atteso è stato quello di mettere in pratica, attraverso la scrittura di un'applicazione distribuita, tutti i concetti espressi nella documentazione ufficiale di Lagom e relativi agli argomenti teorici che vi sono alla base. Per motivi di tempo non ho potuto mettere in pratica alcune questioni a mio avviso importanti, ma posso affermare di aver portato a termine la stesura delle parti core.

3.1 Scenarios

Prendendo spunto dalle features messe a disposizione dalla famosa app di messaggistica "Slack", l'idea di base è stata quella di creare un'applicazione di chat con la quale l'utente può interagire per scambiare messaggi con altri utenti registrati in specifiche aree di discussione, chiamate "Channels".

Un client qualsiasi può effettuare il login e logout, creare un nuovo *Channel* di discussione, entrare a far parte di un canale preesistente o appena creato e scambiare messaggi all'interno di quel canale. Di seguito sono riportate delle Q/A per meglio comprendere in generale il lavoro svolto:

- *Il progetto presenta un'interfaccia grafica con la quale l'utente può interagire?*

  No, il focus non era creare un'applicazione finita ed usabile dall'utente finale, ma anzi di implementare dei microservizi in Lagom con i quali poter interagire simulando la presenza di un'applicazione client.

- *Quali strumenti sono stati usati per interagire con i microservizi creati?*

  Principalmente *Postman* ma anche un add-on di Chrome per effettuare la sottoscrizione ad un topic tramite Web-Socket.

- *Quali feature di chat sono state sviluppate?*

  Il login e logout utente, la creazione di un canale e la sottoscrizione ad esso per poter interagire con gli altri utenti connessi e quindi la possibilità di spedire messaggi. Manca la possibilità di cancellare un canale, verificare che un canale creato non abbia lo stesso nome di uno preesistente, creare un profilo utente, e tutte le altre feature tipiche delle app di messaggistica. L'importante in questo contesto credo sia l'aver testato il framework, al di là del prodotto finito.

- *Quali componenti di Lagom hai implementato?*

  Event Sourcing per la comunicazione e la persistenza dei dati, persistenza tramite database Cassandra, persistenza tramite relational database PostgreSQL e JPA, esposizione di un microservizio tramite chiamate ReST, interazione tra due microservizi per mezzo di Message Broker Kafka e modello publish-subscribe legato all'Event Sourcing, comunicazione intra-servizio tramite pattern publish-subscribe, uso di librerie per il testing dei microservizi (integration testing) e delle Entity create e basate su Event Sourcing.

- *Quale parte dell'implementazione ha impliegato più tempo e studio?*

  Senza dubbio l'implementazione dell'EventSourcing e CQRS. Ciò che è stato detto in precedenza riguardo i tempi e la difficoltà di implementazione è tutto vero ed è stato sperimentato personalmente. Generalmente, anche grazie agli studi universitari, si è abiutati a ragionare con un approccio più relazionale e statico, sia nell'interazione con una base di dati sia nell'architettura software che si va a costruire (ad esempio CRUD). Event Sourcing e CQRS diventano di facile comprensione soltanto dopo che si è realmente compreso il meccanismo interno e come le varie parti interagiscono tra di loro.

- *Quanto è attiva la community di Lagom? E' stata di aiuto per lo sviluppo?*

  Non è stato semplice risolvere i problemi riscontrati, spesso bloccanti, proprio per mancanza di documentazione e di una comunità attiva. La documentazione ufficiale spiega bene i concetti di base ma non cita alcuni dettagli che andrebbero inseriti per facilitare la vita allo sviluppatore. Si è notato che le maggiori interazioni su Stack Overflow e altri siti riguardanti eccezioni o errori legati a Lagom risalivano al 2018 o prima. Credo che Lagom è e rimarrà un framework di nicchia, utilissimo in contesti applicativi dove è già presente un'infrastruttura interna basata su Akka e sul framework Play e dove il team ha una forte conoscenza di Scala.

- *Perché Java e non Scala?*

  Nella community la maggior parte degli sviluppatori consigliano l'uso di Scala per la scrittura dei microservizi in Lagom. La motivazione è senza dubbio legata alla possibilità di scrivere meno codice, alla maggiore versatilità di Scala nei sistemi distribuiti e al fatto che i layer sottostanti (primo tra tutti Akka) sono stati scritti in Scala. Oltre che essere fuori dallo scope del progetto e dell'esame, personalmente ho deciso di scrivere il progetto in Java per essere più efficiente in termini di tempo:

non conoscendo approfonditamente il linguaggio Scala avrei dovuto dedicare del tempo soltanto ulteriore per la comprensione della sintassi e per la scrittura delle funzioni, qualità non importanti in questo contesto.

- *Che cosa ti ha comportato l'aver scelto Java?*

  Il dover utilizzare la libreria Immutables per la creazione di POJOs immutabili, che è comunque risultata di aiuto per evitare di scrivere troppo codice boiler plate.

4 Requirements Analysis

Di seguito vengono riportati alcuni requisiti funzionali e non funzionali di Lagom Chat:

- L'utente deve poter effettuare il login e logout
- L'utente che effettua il login deve avere uno username univoco
- L'utente deve poter creare e/o entrare a far parte di un canale di discussione (Channel)
- L'utente deve poter inviare messaggi ad un canale nel quale si è iscritto
- L' utente può spedire messaggi ad un canale soltanto una volta effettuato il join allo stesso
- L'interazione deve avvenire per mezzo di chiamate ReST a uno o più microservizi che interagiscono tra di loro
- Per il login utente non è necessario avere un meccanismo di autenticazione o altre forme di sicurezza
- Le varie feature devono essere ben distinte secondo una devisione del contesto ben definita, quindi attraverso la creazione di un microservizio ad hoc per ogni funzionalità
- La tecnologia utilizzata per la scrittura dei microservizi deve essere Lagom Framework
- Non è importante la facilità di utilizzo da parte dell'utente finale, quanto l'esplorazione delle varie feature messe a disposizione dal framework

4.1 Terminologia

Per fare chiarezza, in questo contesto per "Utente" si intende un qualsiasi client frontend che fa uso delle interfacce ReST o WebSocket messe a disposizione dai microservizi creati. Durante lo sviluppo si è scelto di utilizzare Postman per il mocking delle interazioni che un utente può svolgere verso i microservizi. Un Channel è invece una room delle più comuni applicazioni di Chat. Il nome prende spunto dall'applicazione *Slack* e servono per suddividere le conversazioni in vari argomenti di discussione.

5 Design

E' stato scelto di suddividere i domini in tre aree contestuali differenti, rappresentate da tre microservizi:

- *User Service*: espone le API per permettere all'utente di aggiungere il proprio username al database utenti in caso di login, o cancellarsi in caso di logout
- *Channel Service*: espone le API per la creazione e l'update di un Canale di discussione. Per update si intende la possibilità di cambiare nome ad un Canale o entrare a farne parte: un utente può aggiungersi alla lista di utenti connessi ad un determinato Canale.
- *Message-Dispatcher Service*: detiene la sola responsabilità di ascoltare i cambiamenti effettuati dal Channel Service e quindi di creare messaggi di log con una semantica. Rimane in ascolto su eventi come la creazione di un canale, l'update di un Channel e l'arrivo di un messaggio da parte di un utente su un determinato Channel.

5.1 Structure

Di seguito vengono riportate le definizioni delle API esposte dai tre microservizi sopra citati. Per una consultazione più approfondita sono stati caricati i sorgenti di Swagger sulla root del progetto, così come anche le collection di Postman per effettuare le chiamate ed interrogare i microservizi.

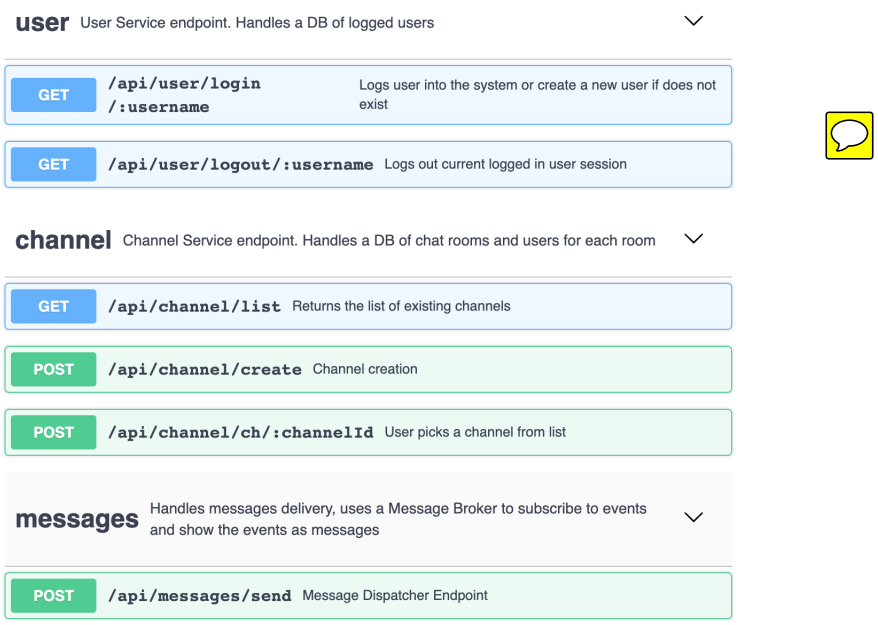

Figure 10: Microservices API calls definition

Va evidenziato l'oggetto rappresentante un Channel, il quale oltre ad avere un identificativo univoco ed un nome, detiene anche una lista rappresentante la lista degli utenti facenti parte di un canale in un dato momento. La call `/api/channel/ch/channelId` rappresenta l'update di un canale preesistente; viene passato tramite chiamata POST un oggetto contenente il nuovo nome del canale (che può essere lo stesso del precedente) e una lista modificata degli utenti attualmente connessi. Non vi è ovviamente alcun meccanismo di sicurezza legato all'aggiunta o rimozione di utenti della lista, ma il comportamento normale sarebbe che a tale lista un utente intento a partecipare a quella room aggiunga soltanto il proprio username.

```
[
    {
        "id": "a960cba5-8458-4d8b-86a4-c0a8875ca010",
        "name": "Topic1",
        "users": ["username1", "username2", "username3"]
    },

    {
        "id": "b9120cgd2-735a-1n3e-75b3-u2a0703ac140",
        "name": "Topic2",
        "users": ["username3","username4", "username5"]
    }
]
```

Listing 4: Channel Object

5.2 Interaction

In questa sezione vengono descritte tramite alcuni diagrammi di Sequenza le principali interazioni tra i microservizi.

La prima figura mostra il login utente e il join ad un canale preesistente. Come si può evincere dal diagramma di sequenza, un nuovo utente contatta lo User Service per effettuare il login. Il login non è altro che l'inserimento di una nuova row nel database contenente lo *username* dell'utente e un id generato in maniera random. Una volta effettuato il login, lo User Service contatta tramite chiamata ReST il ChannelService per avere la lista dei canali attuale esistenti. Si immagini un'interfaccia Web o Mobile dove una volta inserito lo username all'utente viene restituita una lista di stringhe rappresentanti i canali e dove è possibile sceglierne uno per poter proseguire nell'interazione con l'App. La scelta di un canale comporta l'inserimento dello username dell'utente nella di utenti presenti in quel Channel.

Nella figura non è riportato, ma nulla vieta all'utente anche di cambiare *nome* al canale al quale vuole partecipare oltre che aggiungersi alla lista. In altre parole, il nome `joinChannel` può essere fuorviante, ma va detto che oltre che dare la possibilità di aggiungere un nuovo utente alla lista degli utenti connessi ad un canale, può anche cambiarne il nome. Aggiungere un utente alla lista di utenti "connessi" al canale o cambiarne il nome si tradurrà nella creazione di un nuovo evento che verrà reso persistente nel database integrato Cassandra e che al contempo verrà spedito al Message-

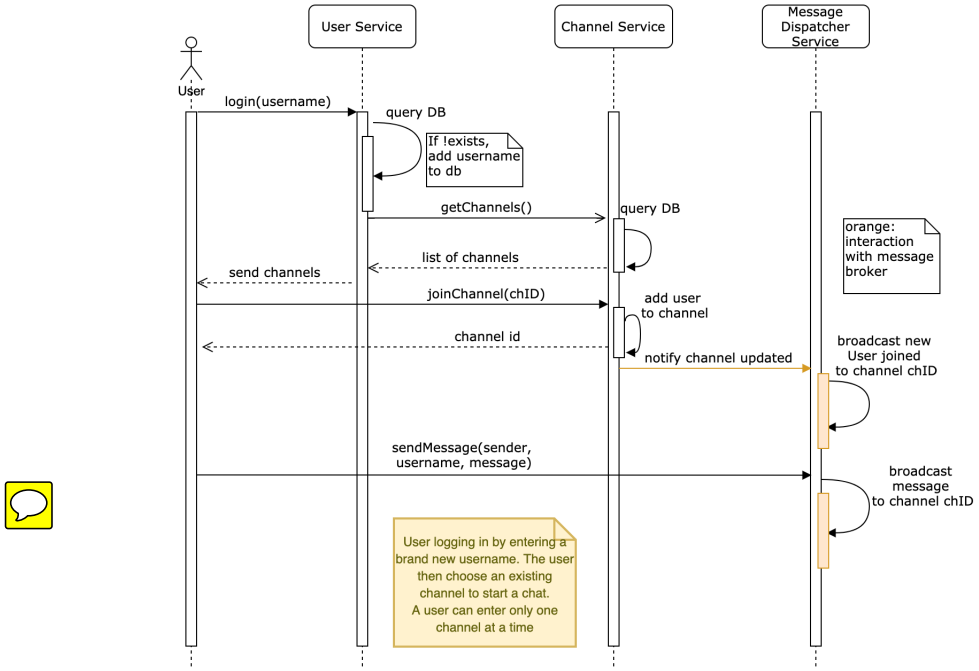

Figure 11: Update or Join Channel Sequence Diagram

Dispatcher, il quale si occuperà di mostrarlo a tutti gli utenti in ascolto. La freccia arancione infatti indica uno scambio di messaggi attraverso Message Broker: l'evento scatenato dall'aggiunta di un utente ad un canale preesistente comporta quindi l'invio di una notifica ad un certo topic (`updated-channels`). Il Message-Dispatcher è un microservizio che resta in ascolto di questo ed altri topic. Una volta che riceve la notifica di canale modificato, mostrerà (molto banalmente, attraverso dei log, ma potrebbe essere anche un altro client web connesso) la nuova lista di utenti connessi al canale o il nuovo nome del canale stesso, cosicché tutti gli utenti connessi a quel canale possano ricevere un avviso. Per capirci, si è voluto prendere spunto dalle chat attualmente in voga: una volta che un nuovo utente viene aggiunto ad un gruppo di persone (diciamo, ad esempio, un gruppo di Telegram), tutti gli utenti facenti parte dello stesso ricevono un avviso riguardante l'evento. Stesso meccanismo avviene per il cambio del nome del gruppo. Infine, l'utente può spedire un messaggio ad uno specifico gruppo, interagendo direttamente con il Message-Dispatcher service, senza dover passare per il Channel Service, che rimarrà così più "scarico" in termini di overhead. L'interazione avviene tramite chiamata ReST in questo caso (freccia nera), ma nulla vieta di utilizzare un approccio publish-subscribe avente ad esempio come topic il nome del Channel. L'oggetto json rappresentante il messaggio, come si evince anche dalle definizioni delle call di Swagger, è costituito da un *sender*, da un *channel* e da un *message*. Il sender è lo username dell'utente che è intenzionato a spedire il messaggio, il channel rappresenta l'id del canale al quale si vuole mandare del testo, e il message è il corpo del messaggio.

Ovviamente prima di fare join ad un Channel occorre crearlo. Il Channel Service mette

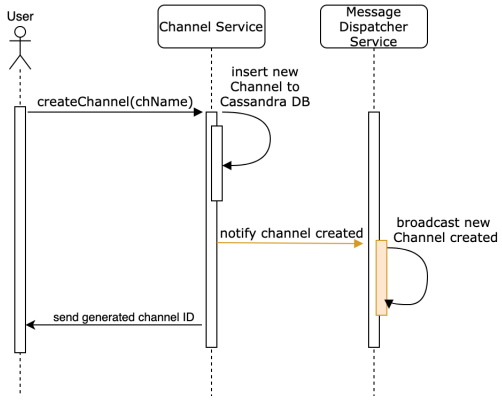

Figure 12: Create Channel Sequence Diagram

a disposizione un'interfaccia per la creazione di un Canale, interfaccia alla quale occorre passare al più il nome. Il Channel Service notificherà al topic *created-channels* l'evento di creazione, il quale verrà ascoltato dal Message-Dispatcher, che notificherà tramite log il nuovo evento. Infine, all'utente viene restituito lo UUID del canale creato, che potrà essere utilizzato successivamente per il effettuare il join e per inviarvi messaggi. Si noti che si parla di eventi perché è il Channel Service ad avere internamente implementato l'Event Sourcing, ma ciò verrà descritto più in dettaglio in seguito.

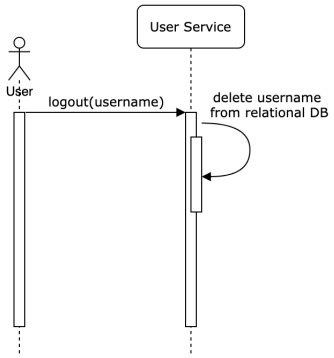

Figure 13: Logout Sequence Diagram

Il logout utente è molto minimale, si tratta di una chiamata al solo User Service, il quale si occuperà di cancellare dal proprio database interno lo username passato come argomento. Le operazioni interne da far svolgere al microservizio a seguito della richiesta di logout potrebbeero essere complicate a piacere. Ad esempio si potrebbe contattare il Channel Service il quale effettuerà un nuovo update di tutti i canali contenenti quello username per cancellarlo dal proprio database, e quindi notificare gli ascoltatori attivi tramite Message-Dispatcher. Questa parte, che in altre parole rappresenta la "Delete" dal punto di vista delle operazioni CRUD, non è stata implementata per mancanza di tempo. La situazione attuale crea ovviamente inconsistenza in quanto, nel database

relazionale degli utenti lo username viene cancellato, mentre nel database dei Channel quello stesso username ora non più esistente, continua ad esistere all'interno dei canali a cui aveva partecipato in un momento precedente. Personalmente credo sia uno degli aspetti negativi dell'avere database separati per ogni microservizio: se un cambiamento avviene in uno e l'altro ne risente, a cascata occorre notificare l'altro dei cambiamenti. Al di là di questa inconsistenza comunque, esempi importanti di comunicazione interna tra microservizi (sia usando ReST che pub-sub) si possono trovare nella fase di login precedentemente descritta, dove l'utente loggato chiederà la lista di canali attivi, o anche nel notificare ad un altro microservizio che un nuovo canale è stato creato o modificato.

5.3 Behaviour

Tra le varie parti del progetto spicca sicuramente quella del Channel Service, nel quale è implementato l'Event Sourcing. Vale la pena quindi entrare più nel merito del comportamento di questo particolare microservizio.

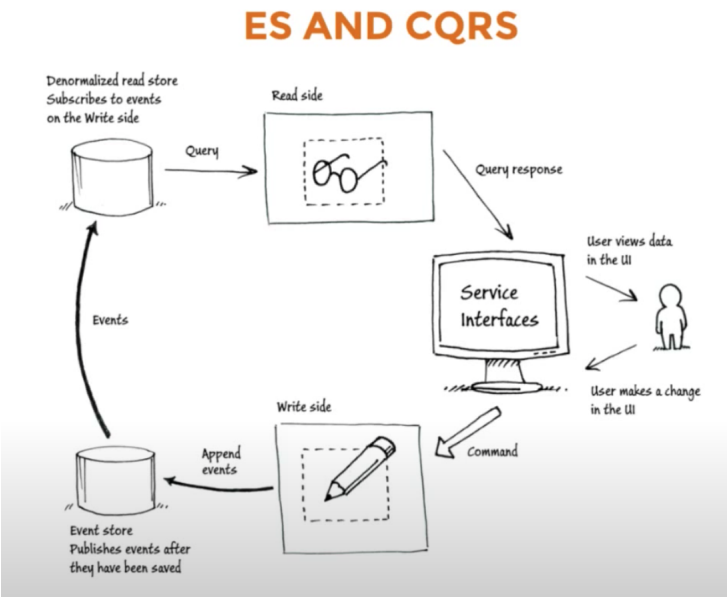

Figure 14: ES and CQRS

Come già discusso in precedenza, CQRS divide il suo persistent layer in Write-Side e Read-Side, le quali possono scalare autonomamente in produzione. Si vuole prendere l'esempio della creazione di un canale da parte dell'utente. Ad alto livello accade che il service interrogato dall'utente internamente spedisce un Command al Write-Side, il quale viene processato. Qui entra in gioco l'Event Sourcing: ogni Command viene convertito in un evento attraverso multipli handler registrati per gestire gli eventi. Tra gli event handler il più importante è quello che si occupa di andare a caccia degli eventi creati e una volta che se ne trovano di nuovi, il Read-Side viene aggiornato.

Qui si introduce l'Eventual Consistency: gli events vengono inseriti all'interno

dell'EventStore ed è soltanto quando gli handler fanno pull degli eventi che il Read-Side viene aggiornato, creando quindi un effettivo delay tra la creazione e la query di un nuovo canale.

Più in dettaglio, il flow è il seguente: un utente sta cerando di creare un nuovo canale. La richiesta di creazione del nuovo canale viene spedita al nostro microservizio (in questo caso tramite ReST al ChannelService). Il microservizio in questione la richiesta di creazione di un nuovo Channel viene convertita in un Command (`CreateChannelCommand`) il quale viene spedito al nostro PersistentEntity. Lagom mette a disposizione il Persistent Entity, il quale corrisponde esattamente al concetto di Aggregate Root in termini Domain-Driven Design. Tale oggetto contiene un handler atto a rendere persistente il Command al posto nostro e come risultato dovrebbe creare un nuovo Channel e metterlo a disposizione in memoria Ram. Allo stesso tempo però, dovrà creare un nuovo evento che descriva tale creazione, quindi creerà un `ChannelCreatedEvent` che verrà salvato all'interno dell'Event Store.

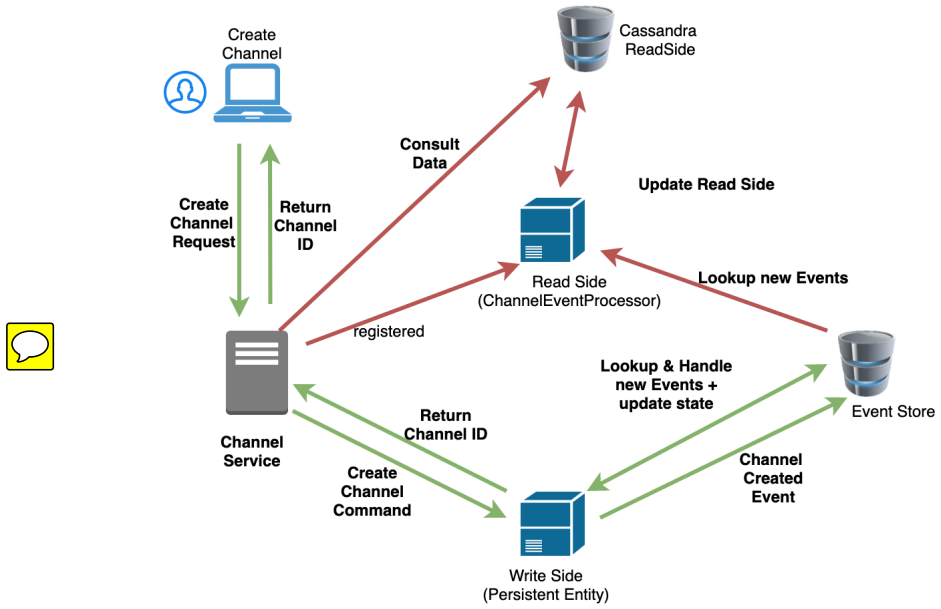

Figure 15: Channel creation request flow

Nel Persistent Entity è possibile anche registrare degli handler che rimangano in ascolto della creazione di tali eventi. Ad esempio, adesso che il Canale di discussione è stato creato, abbiamo la possibilità di aggiornare lo stato (State). Una volta che il Command è stato processato correttamente possiamo ritornare qualunque cosa all'utente, un 200 Ok basterebbe ma potremmo tornare qualsiasi oggetto semanticamente corretto. Nel nostro caso all'utente ritorniamo l'Id del nuovo canale creato. Questo conclude il flow della richiesta dell'utente al Service di creare un nuovo Canale di Chat.

Dietro le quinte, l'Event Sourcing entra di nuovo in azione, ed è tutta qui la reale

differenza rispetto agli approcci tradizionali: i nuovi eventi in arrivo vengono automaticamente caricati nel Read-Side attraverso un processor che rimane in ascolto. Una volta salvati nel Read-Side (rappresentato, nel nostro caso, da un database Cassandra), i dati generati possono essere cercati dall'utente attraverso una semplice chiamata ReST (o di altro tipo) al Servizio. Anche nel caso di questo piccolo progetto ho potuto constatare come un evento creato è disponibile alla lettura soltanto dopo un certo lasso di tempo.

In altre parole, è possibile vedere l'Eventual Consistency in azione semplicemente facendo una call all'API che crea un nuovo canale (CreateChannel in Postman). Gli eventi generati dal `CreateChannelCommand` verranno resi persistenti nel Read-Side Cassandra e saranno quindi disponibili alla lettura prima o poi, dopo tempo non definito a priori. Quindi, chiamando l'API per avere tutti i canali disponibili (GetChannels in Postman) subito dopo la creazione, inizialmente non verrà mostrato nessun nuovo Canale. Dopo qualche secondo il Read-Side viene aggiornato, quindi soltanto ripetendo la chiamata GetChannels in un momento successivo il canale precedentemente creato verrà mostrato nell'oggetto di response con i dati aggiornati.

Non è stato trattato in questa sede, ma per completezza aggiungo che se si volesse invece cancellare un Channel creato, a livello di database potrebbe esserci un effettiva cancellazione della row richiesta, ma a livello ES ciò si traduce in un ulteriore evento che rappresenti la cancellazione il quale verrà impilato come uno stack al di sopra degli altri eventi e successivamente letto dal Read-Side. Inutile dire che questo approccio è rivoluzionario rispetto a quelli tradizionali, poiché vi è una divisione netta di tutto ciò che riguarda la scrittura dalla lettura e nulla vieta di spostare le due parti su nodi fisici differenti anche dislocati geograficamente e replicati in più Cluster (altra implementazione presente in Lagom ma non trattata) per garantire maggiore consistenza. Con l'inserimento dell'Event Sourcing in Lagom e con le altre tecniche di clustering che il Framework mette a disposizione, vi è a mio avviso un tentativo (forse riuscito) di soddisfare le tre punte del triangolo del *CAP Theorem*. Grazie a questi meccanismi si garantisce maggiore Consistenza, ma anche Disponibilità (purtroppo non real time ma comunque eventually) e la separazione dei concetti porta anche a risolvere in parte la possibilità di avere perdite di connessione tra due nodi, proprio perché meno carichi di richieste da processare. Ovviamente anche la rete sottostante dovrà fare la sua parte e su questo aspetto si ha meno potere.

6 Implementation Details

In questa sezione vengono descritte le principali classi del progetto Lagom Chat e come avvengono le interazioni più in dettaglio, con particolare attenzione verso il Channel Service, considerato il più importante dal punto di vista del scope del progetto.

Nel package `it.unibo.channel.api` l'interfaccia più importante è senza dubbio `ChannelService`, la quale descrive le API che il servizio espone e i topic al quale esso fa riferimento per spedire messaggi. Le classi `AbstractCreateChannelRequest` e `AbstractCreateChannelResponse`, insieme al Pojo `AbstractChannel`, descrivono invece i dati che vanno scambiati come richieste e risposte nelle call relative alla creazione

e all'update di un Channel. `ChannelEvent` rappresenta invece la classe che descrive l'evento generato dal Command di creazione del canale ma anche di update.

Nel package `it.unibo.channel.impl` invece vi sono le classi che fanno uso delle interfacce sopra descritte. Molto importante citare la classe `ChannelEntity` che rappresenta l'event sourced aggregate ed è quindi una implementazione del Persistent Entity in Lagom [8]. Ha bisogno di un Command, uno State e un Event e descrive i vari handler per la gestione dei Command che verranno tradotti in eventi.

Ad esempio, dall'implementazione del metodo `createChannel()` in `ChannelServiceImpl` si può intuire che viene chiesto al PersistentEntity di creare un CreateChannel Command. Tale richiesta verrà presa a carico dall'handler cha ha come argomento per l'appunto CreateChannel. L'handler si occuperà di creare un evento ChannelCreated e quindi di rendere persistenti (tramite il metodo `thenPersist()`) i cambiamenti nel Write-Side e ritornare l'Id del Canale creato. La callback contenente l'id verrà ancora una volta gestita internamente dal persistent entity e quindi restituita al metodo `createChannel()` il quale restituisce all'utente la risposta in Json contenente l'Id.

```
//ChannelServiceImpl class
@Override
public ServiceCall<CreateChannelRequest, CreateChannelResponse>
    createChannel() {
    return request -> {
        logger.info("Creating a brand new Channel: {}", request);
        UUID uuid = UUID.randomUUID();
        return convertErrors(persistentEntities
                .refFor(ChannelEntity.class, uuid.toString())
                .ask(CreateChannel.of(request)));
    };
}
[...]
//ChannelEntity class
b.setCommandHandler(CreateChannel.class, (cmd, ctx) -> {
    if (state().getChannel().isPresent()) {
        ctx.invalidCommand(String.format("Channel %s is already created",
    entityId()));
        return ctx.done();
    } else {
        Channel channelModel = Channel.of(entityId(),
                cmd.getCreateChannelRequest().getName(),
                cmd.getCreateChannelRequest().getUsers());
        final ChannelCreated channelCreated = ChannelCreated.builder().
    channel(channelModel).build();
        return ctx.thenPersist(channelCreated, evt ->
                ctx.reply(CreateChannelResponse.of(entityId())));
    }
});
```

Listing 5: Persistent Entity

Utile notare come nella classe `ChannelCommand` vanga definito un `PersistentEntity.ReplyType<CreateChannelResponse>`. Il tipo di ritorno viene esplicitamente dichiarato nel command e quindi anche l'interfaccia del metodo

`createChannel()` dovrà avere lo stesso tipo di ritorno. In questo caso l'id del canale creato viene wrappato all'interno dell'oggetto `CreateChannelResponse`.

Si noti come tale oggetto non sia presente tra i due package citati, ma viene automaticamente generato attraverso lo scaffolding della libreria *Immutables*. Alla prima apertura del progetto si potrebbero incorrere a molti errori di classe non trovata. Questo perché Immutables va correttamente configurato affinché generi attraverso le classi astratte definite nei package le classi reali che andranno ad essere utilizzate, ad esempio dall'implementazione del ChannelService.

Altro elemento sicuramente da citare è il `ChannelEventProcessor`, classe che si occupa della creazione del db e delle tables in Cassandra in maniera del tutto automatica, ma anche di rendere persistente un evento, come ad esempio nel caso di `processChannelUpdated()`. Si noti che nel file *build.sbt* vi è una riga

(`lagomCassandraCleanOnStart`) che dichiara esplicitamente che tutti i database e i dati salvati in Cassandra devono essere cancellati fino a prossimo avvio del programma. Questo è molto utile per fini di sviluppo, così da avere un database pulito ad ogni nuovo run. Sicuramente molto pericoloso e da evitare in produzione.

Piccola nota sulle best practice: è utile sottolineare la scelta dei nomi. Si può notare infatti che un comando ha come parola finale Command ed è spesso imperativa, un evento creato ha come verbo un verbo al passato (i.e. ChannelCreated).

Fin ora si è parlato di una possibile implementazione sotto forma di EventSourcing di una API che un servizio può esporre, ma è utile anche spendere qualche parola sul Message Broker fornito da Lagom (libreria `lagomJavadslKafkaBroker`), sotto al quale gira un server Kafka pronto a ricevere i messaggi pubblicati su un certo topic ma anche a mantenere in memoria i messaggi in caso di problemi di rete, per poi spedirli una volta che torna disponibile.

Se da una parte abbiamo un handler di un Command impartito da un utente, lo tramuta in evento e lo rende persistente, dall'altra abbiamo un produttore di messaggi che rimane in ascolto e appena nota un nuovo evento lo pubblica su un certo topic, creando così una forma di comunicazione alternativa. Per fare un esempio, vorrei focalizzare l'attenzione su `updatedChannelTopic()`, il quale tramite un **TopicProducer** fornito da Lagom, si occupa di pubblicare l'evento di ChannelUpdated sul topic "updated-channels". Questa volta sarà il Message Dispatcher, che nella classe `ChatMessageEventsSubscriber` implementa un sottoscrittore ai topic generati dall'altro microservizio, tra cui anche quello relativo al cambio di stato di un Canale (sia esso un cambio di nome o un join di un utente allo stesso). In Lagom i topic sono strongly typed, questo permette una migliore ingegnerizzazione preventiva dell'utilizzo del dato poiché il sottoscrittore conosce a priori di che tipo sarà il dato ricevuto e quindi cosa contiene. [9]. Inoltre occorre avere un riferimento esplicito ad un microservizio per potersi sottoscrivere ad uno dei suoi topic. Il riferimento lo si può ottenere tramite Dependency Injection, come si può evincere dalla classe appena citata.

```
channelService.updatedChannelTopic()
    .subscribe()
    .atLeastOnce(Flow.fromFunction((ChannelEvent event) -> {
                logger.info("CHAT MESSAGE EVENT: Channel updated  " + event
    );
                return Done.getInstance();
            })
    );
```

Listing 6: Example of Message Broker API subscriber

Sempre basata sul pattern publish-subscribe ma leggermente diversa è invece la libreria `lagomJavadslPubSub` che permette a cli pubblica di spedire messaggi ad un certo topic ma senza conoscere chi saranno i sottoscrittori, così come questi ultimi non hanno idea di chi è il mittente. Questa è la prima grande differenza rispetto alla libreria sopra citata. [10] Un'altra differenza da far notare è che quest'ultima va utilizzata generalmente per una comunicazione di tipo pub-sub soltanto intra-servizio, mentre Message Broker API è stata studiata per la comunicazione tra microservizi differenti. Inoltre JavadslPubSub non fa uso degli eventi generati dall'Event Sourcing per spedire messaggi, ma si limita ad implementare un semplice sender e receiver e deve essere un'entità esterna (come ad esempio un client e quindi l'utente) a pubblicare i messaggi.

Ho voluto utilizzare questa libreria per la spedizione e il recapito dei messaggi su un certo Channel. Nella classe `MessageServiceImpl` viene implementata una chiamata ReST per l'invio di un messaggio. Se l'utente usa quella API, il microservizio si occupa di pubblicare il messaggio richiesto. Lo stesso microservizio contiene un subscriber (metodo `getPublishedMessageStream()`) il quale banalmente stampa un log del messaggio (ma in una implementazione futura potrebbe essere un client web che si sottoscrive tramite web-socket e riceve il messaggio spedito da un altro utente).

Di seguito sono stati riportati gli esempi di messaggi ricevuti dai subscriber sia utilizzando l'una che l'altra libreria.

```
2020-08-08 08:52:55,521 INFO  it.unibo.message.dispatcher.impl.MessageServiceImpl - CHAT MESSAGE Event:
           new Channel created  ChannelCreated{id=f604a1a7-2510-4a3b-a708-683601adbbb1, name='Food', users=[]}
2020-08-08 08:53:49,172 INFO  it.unibo.message.dispatcher.impl.MessageServiceImpl - CHAT MESSAGE Event:
           new Channel created  ChannelCreated{id=ecb6513b-d711-455f-a5e2-b8c01459d48c, name='Technology', users=[]}

2020-08-10 21:00:19,655 INFO  it.unibo.message.dispatcher.impl.MessageServiceImpl - CHAT MESSAGE EVENT:
         Channel updated  ChannelUpdated{id=ecb6513b-d711-455f-a5e2-b8c01459d48c, name='Technology', users=[username1]}
```

Figure 16: Message Broker API subscriber logs

Nella prima figura vengono riportati i messaggi di log generati dal Message Dispatcher Service nel sottoscriversi agli eventi generati dalla creazione e modifica di uno o più Channel.

Nella seconda figura invece, ho voluto utilizzare una extension di Google Chrome che funge da client di Web Socket per poter ascoltare i messaggi del subscriber `getPublishedMessageStream()`, stampati a seguito della service call `publishMessage()`, una chiamata POST effettuata tramite Postman contenente il messaggio da inviare con

Figure 17: JavadslPubSub subscriber stream logs

tanto di Channel di destinazione.

Per completezza aggiungo che un altro metodo di comunicazione tra microservizi è quello di utilizzare le chiamate ReST. Un esempio da citare è quello del login: l'utente contatta lo User Service che si prenderà a carico la richiesta di inserire un nuovo utente nel database relazionale. Se non ci sono username duplicati e tale richiesta è andata a buon fine, lo User Service contatta il Channel Service per avere subito una lista dei Channel disponibili. Questa feature è stata implementata immaginando una UI dove l'utente, una volta loggato ha subito a disposizione la possibilità di fare join ad un canale scegliendolo dalla lista e iniziare a chattare. L'invocazione di un metodo esposto dall'interfaccia di un microservizio B da parte di un microservizio A avviene attraverso il metodo asincrono `invoke()`, messo a disposizione dalle classi Lagom che restituisce una Future alla quale si può sottoscrivere attraverso un `thenApply(function -> {})`.

Un esempio di questa chiamata si può notare nel metodo `login()` della classe `UserServiceImpl`, come anche nelle classi dedicate al testing dei microservizi.

7 Run Instructions

Come scritto nel file di Readme, è possibile avviare il progetto tramite un semplice comando: `sbt runAll` Le condizioni necessarie affinché tutto funzioni senza errori di compilazione sono le seguenti:

- Avere correttamente installato `sbt`
- Avere correttamente installato nativamente PostgreSQL e creato manualmente uno schema chiamato `user_db` avente come username e password la parola `postgres`, aver creato una table chiamata `usermodel` contenente due colonne, `id` (varchar e anche chiave primaria) e `username` (varchar). Sfortunatamente Lagom non permette la creazione interna automatica del database relazionale, come invece succede per il database NoSql Cassandra. Per cambiare le impostazioni di connessione al database locale si può fare riferimento al file `application.conf` all'interno della cartella resources nel modulo UserImpl.

Di seguito riporto uno screenshot della connessione tramite IDE al db creato localmente.

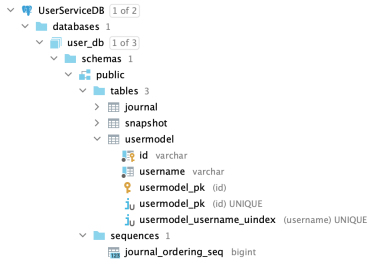

Figure 18: PostgreSQL User Service setup

- Avere correttamente configurato nel proprio IDE (fortemente consigliato IntelliJ) la libreria Immutables, per avere una cartella `target` nella quale generare le classi. Di seguito riporto uno screenshot della configurazione che è stata utilizzata per la generazione automatica delle classi, tuttavia consiglio di consultare la guida ufficiale presente in bibliografia [6] per un setup ad hoc.

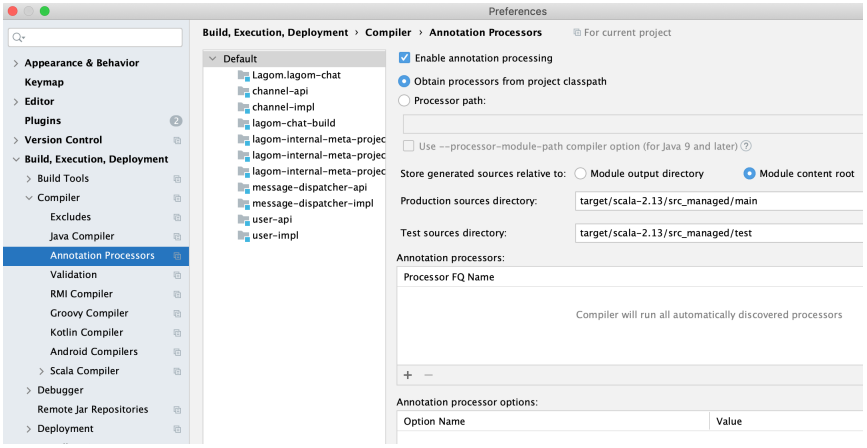

Figure 19: Immutables IDE setup

- Avere correttamente installato Java 8 e il plugin Scala

8 Usage Examples

Esempio tipico di utilizzo è tramite Postman (importare prima le postman collections fornite) una volta che il progetto è avviato. Ad esempio, una volta loggato un nuovo utente, effettuare una query al db PostgreSQL per verificare che sia stato effettivamente inserito. Oppure, una volta creato un canale tramite la call CreateChannel di Postman, attendere qualche secondo e poi effettuare il fetching dei Channels creati tramite la call GetChannels di Postman.

9 Conclusions

In conclusione posso affermare che non è stato semplice interagire con Lagom e di certo la mancanza di una comunità attiva ha influito molto sui tempi e la qualità del software stesso. Detto ciò ritengo sia un framework molto potente (anche se poco apprezzato) e che valga la pena di studiare ed utilizzare, soprattutto in ambito accademico. Ho trovato particolari difficoltà nel mettere in pratica (tramutare in codice) i concetti teorici legati al pattern Event Sourcing e CQRS: per quanto ritenga sia una svolta a livello architetturale e tecnologico e che porti benefici non indifferenti in produzione, credo che siano ancora pochi i contesti aziendali disposti ad utilizzare questa tecnologia, soprattutto per un aspetto legato alla formazione degli sviluppatori, ai tempi di sviluppo e ai difetti e i bug spesso non facilmente rintracciabili che ne possono derivare.

9.1 Future Works

Sono molte le feature mancanti se si volesse implementare una applicazione di chat completa. Ad esempio, come già citato, un utente che effettua il logout dovrebbe essere cancellato a cascata anche da tutti i canali di discussione a cui aveva partecipato. Parte essenziale sicuramente mancante è la UI, quindi una interfaccia grafica Web che oltre che permettere all'utente di loggarsi e creare un canale, resti in ascolto dei messaggi inviati dal Broker tramite websocket per ricevere nuovi messaggi di chat o avvisi di cambio di stato di un Canale (nuovo utente entrato in un canale, cambio di nome di un canale, nuovo canale creato). Sempre per motivi di tempo, non ho potuto indagare su tutti gli aspetti legati al deployment e la messa in produzione, ma questo si allontana leggermente dal fine del progetto, più legato allo studio della tecnologia e l'investigazione sull'Event Sourcing.

9.2 What did I learn

Aspetto più importante, posso affermare di aver imparato a padroneggiare l'Event Sourcing, sia a livello teorico che pratico tramite un'implementazione in Java. Non meno importante l'aver avuto la preziosissima opportunità di costruire un'applicazione squisitamente distribuita, a Microservizi, e con l'aiuto di un framework sicuramente innovativo e che fornisce ottimi strumenti di supporto utili a facilitare la vita dello sviluppatore. Interessante anche tutto l'aspetto del Testing dei Microservizi, il quale si pone ad un livello di certo superiore rispetto al semplice UnitTesting della singola classe o componente.

References

[1] Jonas Bonér. *Reactive Microservices Architecture - Design Principles for Distributed Systems*. O'Reilly Media, 2016.

[2] Martin Fowler. CQRS. https://martinfowler.com/bliki/CQRS.html, 2011.

[3] Martin Fowler. Microservices. https://martinfowler.com/articles/microservices.html, 2014.

[4] Vladimir Khorikov. Types of CQRS. https://enterprisecraftsmanship.com/posts/types-of-cqrs/, 2015.

[5] Lightbend. Advantages of Event Sourcing. https://www.lagomframework.com/documentation/current/java/ESAdvantage.html.

[6] Lightbend. Immutables Setup. https://www.lagomframework.com/documentation/current/java/ImmutablesInIDEs.html, 2020.

[7] Lightbend. Integrating with Akka. https://www.lagomframework.com/documentation/current/java/Akka.htmll, 2020.

[8] Lightbend. Lagom Persistent Entity. https://www.lagomframework.com/documentation/current/java/PersistentEntity.html, 2020.

[9] Lightbend. Message Broker API. https://www.lagomframework.com/documentation/current/java/MessageBrokerApi.html, 2020.

[10] Lightbend. Publish-Subscribe. https://www.lagomframework.com/documentation/current/java/PubSub.html, 2020.

[11] Lightbend. Service Descriptors. https://www.lagomframework.com/documentation/current/java/ServiceDescriptors.html, 2020.

[12] Rodrigo Botti Medium. My Journey into CQRS and Event Sourcing. https://medium.com/nexa-digital/my-journey-into-cqrs-and-event-sourcing-4bd7d0c1c670, 2019.

[13] Microsoft. CQRS Pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs, 2020.

[14] Saurabh. *Eventual vs Strong Consistency in Distributed Databases*, 2017. https://hackernoon.com/eventual-vs-strong-consistency-in-distributed-databases-282fdad37cf7.

[15] Jonathan West. How the Lagom framework enables scalable, reactive Microservices in Java and Scala. https://medium.com/@jgwest/how-the-lagom-framework-enables-scalable-reactive-microservices-in-java-and-scala-co
2019.