# 

# **ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA CAMPUS DI CESENA**

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

**PROGETTAZIONE E SVILUPPO DI UN SISTEMA DISTRIBUITO DI NOTIFICHE BASATO SU DATABASE REAL-TIME**

*Elaborato in*

BASI DI DATI AVANZATE

*Relatore* *Presentata da*

**Prof. Alessandra Lumini** **Francesco Foschini**

Anno Accademico 2019 - 2020

# Sommario

Lo studio eseguito prevede la realizzazione di un sistema che permetta di distribuire notifiche che si riferiscono a eventi generati sui software di backend e di frontend aziendali, per fornire informazioni costantemente aggiornate agli utenti delle applicazioni web e monitorare lo stato di avanzamento di task/procedure.

Per il design del sistema è stato necessario acquisire una solida conoscenza dei sistemi distribuiti, dei problemi di concorrenza sui dati condivisi e dell’interfacciamento con database real-time.

Il progetto ha previsto l’implementazione del server e della libreria di sistema UtilityRethink. In seguito ad un’attenta ricerca iniziale è stato deciso di affidare il ruolo del server di sistema al dbms RethinkDB. La libreria, invece, è stata scritta in linguaggio C# su piattaforma .NET Core allo scopo di garantire ai servizi client l’interfacciamento al server di sistema. Il codice prodotto è stato organizzato in maniera efficiente e riutilizzabile nei prodotti della società.

Dal punto di vista del database, è stato studiato lo schema migliore di rappresentazione delle notifiche per ottenere le migliori prestazioni possibili a fronte delle necessarie caratteristiche di consistenza e isolamento.

L’attività è stata svolta interamente presso l’azienda Energy Software s.r.l. di Faenza che è un’azienda nata nel 2015 grazie a un team di persone con esperienza decennale unite dall’obiettivo di fornire tutta una serie di servizi legati al mondo dell’informatica tra cui lo sviluppo di software per i mercati energetici, lo sviluppo di applicativi per specifiche esigenze del cliente e la consulenza sistematica specializzata su server e networking.

I risultati ottenuti sono stati oggetto di misurazione e presentazione.

Il seguito della tesi è articolato in 5 capitoli:

* Capitolo 1: introduce nei dettagli la collocazione di questo sistema sviluppato. Sono inizialmente trattati i nuovi sistemi dbms real-time e il design pattern Publish/Subscribe. Vengono infine definite le specifiche funzionali e architetturali del sistema.
* Capitolo 2: presenta altre tecnologie che potevano essere utilizzate per implementare il sistema.
* Capitolo 3: descrive la progettazione della libreria di sistema UtilityRethink implementata.
* Capitolo 4: analizza i software che sono stati scelti e utilizzati per l’implementazione del sistema per il caso di utilizzo aziendale.
* Capitolo 5: mostra come è stato implementato il server del sistema.

**Indice**

[Sommario 3](#_Toc61196289)

[Specifiche e descrizione del problema 7](#_Toc61196290)

[Introduzione al problema 7](#_Toc61196291)

[Database-Realtime 7](#_Toc61196292)

[Publish/Subscribe 8](#_Toc61196293)

[Specifiche architetturali 9](#_Toc61196294)

[Specifiche funzionali 9](#_Toc61196295)

[Altri sistemi esistenti utilizzabili in futuro (Librerie e tecnologie) 10](#_Toc61196296)

[Firebase 10](#_Toc61196297)

[Database in tempo reale FireBase 10](#_Toc61196298)

[Meteor 11](#_Toc61196299)

[Parse 11](#_Toc61196300)

[RabbitMQ 13](#_Toc61196301)

[Conclusioni sulle altre tecnologie utilizzabili 14](#_Toc61196302)

[Scelte architetturali per il caso d’uso aziendale 15](#_Toc61196303)

[Software utilizzato per il server 15](#_Toc61196304)

[RethinkDb 15](#_Toc61196305)

[Docker 19](#_Toc61196306)

[Software utilizzati per l’implementazione della libreria di sistema Utility Rethink 20](#_Toc61196307)

[Metodologia di Progettazione 21](#_Toc61196308)

[Libreria UtilityRethink 22](#_Toc61196309)

[Diagramma dei casi d’uso 22](#_Toc61196310)

[Diagramma delle attività e di sequenza 23](#_Toc61196311)

[Base Dati 25](#_Toc61196312)

[Notifiche 26](#_Toc61196313)

[Notifiche di nuovo dato 28](#_Toc61196314)

[Notifiche di esecuzione 29](#_Toc61196315)

[Diagrammi delle classi 30](#_Toc61196316)

[UtilityRethink 31](#_Toc61196317)

[Simple Injector 32](#_Toc61196318)

[DbManager 33](#_Toc61196319)

[NotificationsManager 34](#_Toc61196320)

[IQueryNotification 35](#_Toc61196321)

[IRXNotifier 36](#_Toc61196322)

[Esempio di OnNext, OnError e OnComplete 38](#_Toc61196323)

[Connection 39](#_Toc61196324)

[Implementazione Server 40](#_Toc61196325)

[Server a singolo nodo 40](#_Toc61196326)

[Server a due nodi 41](#_Toc61196327)

[Server a cinque nodi 42](#_Toc61196328)

[Conclusioni sui tre server implementati 43](#_Toc61196329)

[Comandi per la gestione del server di sistema 44](#_Toc61196330)

[Sharding e Replication 46](#_Toc61196331)

[Gestione con Interfaccia Web 46](#_Toc61196332)

[Gestione con ReQL 48](#_Toc61196333)

[Gestione nella libreria UtilityRethink 49](#_Toc61196334)

[Conclusioni 50](#_Toc61196335)

[Bibliografia 51](#_Toc61196336)

[Ringraziamenti 54](#_Toc61196337)

**Capitolo 1**

# Specifiche e descrizione del problema

# 

## Introduzione al problema

### Database-Realtime

Gli utenti si aspettano un’immediata reattività dalle applicazioni moderne. Si presume, quindi, che qualsiasi modifica apportata nelle applicazioni si rifletta immediatamente in tutte le interfacce di ogni utente in tempo reale [[1]](#database_realtime1).

Lo sviluppo delle applicazioni reattive effettuate con la tecnologia di database tradizionale, tuttavia, presenta delle difficoltà perché i *dbms*, sistemi per l’archiviazione e il recupero dei dati, sono stati sviluppati per decenni attorno a un modello di accesso richiesta-risposta puramente “*pull-based*”.

Rispondendo alla necessità di reattività, si è sviluppata una nuova classe di sistemi *dbms* basati su un’architettura diversa dalla maggior parte dei sistemi *dbms* relazionali e non relazionali preesistenti. Invece di eseguire il *polling* per le modifiche avvenute sui dati di un database, sovraccaricando il sistema, questi nuovi *dbms* sono orientati al “*push delle modifiche”.* Lo sviluppatore, infatti, può richiedere al sistema stesso di inviare continuamente i risultati di *query* alle applicazioni client in tempo reale. Questi sistemi sono spesso definiti *database realtime* poiché mantengono i dati sul client sincronizzati con lo stato corrente del database.

L’architettura *push* in tempo reale dei sistemi *database real-time* facilita quindi lo sviluppo di applicazioni web reattive. La modifica dei dati è intercettata e notificata in tempo reale a tutti i servizi client. L’utilizzo di questi particolari e nuovi sistemi è quindi molto consigliato quando le applicazioni richiedono di essere aggiornate in tempo reale. Siccome, ad esempio, i dati correnti dei mercati della borsa di New York sono estremamente dinamici nel tempo, l’architettura offerta dai *database real-time* potrebbe essere molto vantaggiosa per un’applicazione che disegna i grafici in tempo reale di questi mercati azionari [[2]](#database_realtime2). Potrebbe inoltre essere interessante utilizzare questi sistemi nei contesti di applicazione di progettazione collaborativa, quando un utente, ad esempio, decide di cambiare la posizione di un pulsante. Sarà il *dbms* stesso a informare gli altri utenti, che stanno lavorando contemporaneamente allo stesso progetto, dell’aggiornamento effettuato. Si potrebbe infine utilizzare nello sviluppo di videogiochi multiplayer e nello streaming di applicazioni di analisi.

L’adattamento dei sistemi gestionali di database alle esigenze di aggiornamento in tempo reale rappresenta ancora un’enorme sfida ingegneristica.

Al contrario, però, i nuovi sistemi database real-time non sono sempre la scelta ottimale. Quando si ha la necessità di rispettare le proprietà *ACID* ovvero atomicità, coerenza, isolamento e durabilità dei dati occorre fare affidamento ai classici sistemi *dbms* relazionali e quando si eseguono analisi ad alta intensità di calcolo, occorre utilizzare un sistema di *Big Data* come *Hadoop* o ad un archivio orientato alle colonne come *Vertica*.

### Publish/Subscribe

Il sistema che è stato sviluppato ha richiesto un unico importante e particolare requisito di partenza: il server di sistema stesso deve essere in grado di notificare eventi in real-time ai servizi client interessati. Di conseguenza, il sistema creato è un’implementazione del design pattern *Publish/Subscribe* [[3]](#Publish_Subscribe).

Proprio per questa motivazione la tecnologia scelta per l’implementazione del server di sistema è un *dbms* real-time disponibile sul mercato: *RethinkDB*.

Nello schema *Publish/Subscribe*, il mittente dei messaggi è il server, mentre i destinatari sono i servizi client che dialogano attraverso un tramite, chiamato *dispatcher* o *broker*. Il server (*publisher*) non essendo consapevole dell'identità dei destinatari (detti *subscriber*) si limita a "pubblicare" i propri messaggi al *dispatcher*.

I client si rivolgono a loro volta al *dispatcher* abbonandosi alla ricezione di messaggi.

Il *dispatcher*, quindi, inoltra ogni messaggio inviato dal server solo ai servizi client interessati a quel messaggio specifico.

Il design offerto da *Publish/Subscribe* implica, infine, che per qualsiasi tecnologia del server scelta, non siano noti i servizi client che s’interfacciano a esso, contribuendo di conseguenza alla scalabilità del sistema.

Utilizzando questo pattern di messaggistica i servizi client del sistema riescono a registrarsi alla ricezione di notifiche.

Il meccanismo di sottoscrizione consente, inoltre, ai servizi client di precisare a quali messaggi sono interessati: un client, ad esempio, potrebbe "abbonarsi" solo alla ricezione di notifiche aventi un determinato valore di un campo.

## Specifiche architetturali

Il sistema che è stato implementanto in questo progetto coinvolge, come mostrato in figura 1.1 tre entità software principali: un applicativo client, una libreria e un server. Il servizio client riesce a interagire con il server utilizzando la libreria.

Come requisito di partenza, il sistema, oltre alla capacità di distribuire ai client le notifiche, deve essere in grado di archiviare e salvare le stesse. Il server, per questo motivo, deve essere formato da due sotto-componenti: un dbms per il salvataggio e l’archiviazione delle notifiche e un frame work che garantisca la loro distribuzione a tutti gli applicativi client.

La libreria che è stata implementata e che garantisce ai servizi client di interagire con il server di sistema deve, come requisito, utilizzare l’api offerta dal frame work per garantire a tutti i client la ricezione in real-time delle notifiche.

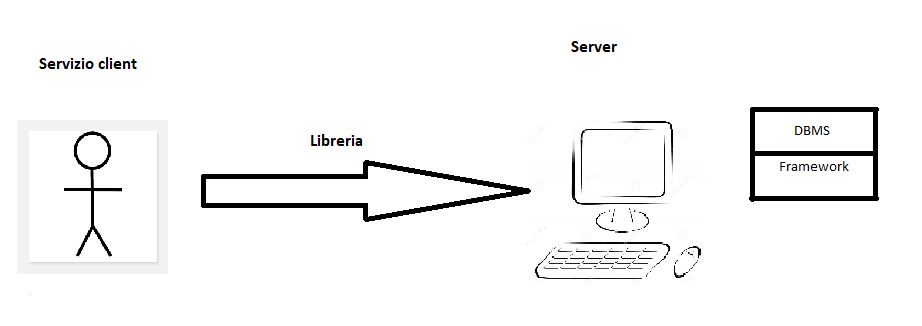


Figura 1.1

## Specifiche funzionali

La libreria di sistema da implementare deve consentire ai servizi client che ne usufruiscono di registrarsi alla ricezione di messaggi a cui si è interessati dal server.

Nel contesto aziendale, in particolare, i servizi di frontend e backend possono registrarsi alla ricezione di notifiche con un certo argomento. Se, ad esempio, è inserita nel database una nuova notifica con argomento uguale a “A” allora saranno avvisati tutti i servizi interessati a notifiche aventi argomento uguale a “A”.

Oltre alla capacità di garantire la distribuzione delle notifiche la libreria permette, inoltre, di eseguire operazioni CRUD sulle tabelle che sono considerate di sistema dalla libreria implementata (es. la tabella che raccoglie le notifiche chiamata *Notifications*).

Il sistema fornisce, infine, funzionalità di riconfigurazione delle politiche di Replication e Sharding attuate sui dati di una tabella e funzionalità gestionali per il database come la creazione e la cancellazione di nuove tabelle e di indici.

**Capitolo 2**

# Altri sistemi esistenti utilizzabili in futuro (Librerie e tecnologie)

Per l’implementazione del sistema, come spiegato nel capitolo 1, potevano essere utilizzate tutte le tecnologie in grado di implementare il design pattern *Publish/Subscribe*, come tutti i sistemi dbms real-time disponibili in commercio: RethinkDB, Firebase, Meteor, Parse. Si è utilizzato per questo progetto il dbms RethinkDB che sarà illustrato in seguito.

## Firebase

Firebase è ora la piattaforma di punta di Google per la creazione di applicazioni per dispositivi mobile e web ed è stato integrato ad altri servizi Google per offrire un insieme più ampio di prodotti per gli sviluppatori. Ad esempio Google cloud messaging, il servizio google per mandare notifiche push ai dispositivi Android, è superato dal prodotto FireBase Cloud Messaging, che ha aggiunto la funzionalità di invio notifiche push sia ai dispositivi iOS sia dispositivi Web [[4]](#Firebase).

Firebase Cloud Messaging (FCM) è una soluzione cloud multipiattaforma che può essere utilizzata gratuitamente; consente agli sviluppatori di applicazioni di terze parti di inviare notifiche o messaggi dai server agli utenti della piattaforma. Questo sistema consente, quindi, all’utente di ricevere messaggi di notifica o di dati che possono essere decifrati dal proprio codice dell’applicazione client. FCM può recapitare messaggi di destinazione alle applicazioni con tre metodi: a un singolo dispositivo, a un gruppo di dispositivi o ai dispositivi che sono iscritti all’argomento del messaggio [[5]](#Firebase_cloud_messaging).

### Database in tempo reale FireBase

Firebase Realtime Database è un database ospitato nel cloud. I dati vengono archiviati come JSON e sincronizzati in tempo reale con tutti i client connessi.

A differenza di altri archivi di dati basati su JSON, un’istanza Firebase non è una raccolta di documenti JSON, ma un unico documento JSON: una struttura ad albero di oggetti ed elenchi nidificati. Per accedere ai dati è necessario navigare attraverso la gerarchia e richiedere specifici nodi figlio per i quali è possibile ricevere aggiornamenti immediati quando altri utenti modificano i dati.

FireBase Realtime Database permette di creare applicazioni collaborative consentendo un accesso sicuro al database direttamente dal codice lato client.

I dati sono conservati localmente e, anche offline, gli eventi in tempo reale continuano a verificarsi, offrendo all’utente finale un’esperienza reattiva. Quando il dispositivo riacquista la connessione, il database in tempo reale sincronizza le modifiche ai dati locali con gli aggiornamenti remoti che si sono verificati mentre il client era offline [[6]](#Firebase_realtime_db).

Il Real-time Database offerto da Firebase è un database NoSQL e come tale ha ottimizzazioni e funzionalità diverse rispetto a un database relazionale. L’API del database in tempo reale è progettata per consentire solo operazioni che possono essere eseguite rapidamente. Ciò consente di creare un’ottima esperienza che può servire milioni di utenti senza compromettere la reattività. Per questo motivo, è importante pensare a come gli utenti devono accedere ai dati e quindi strutturarli di conseguenza.

I dati archiviati in un Firebase Realtime Database sono recuperati collegando un listener asincrono a un riferimento del database. Il listener è attivato una volta per lo stato iniziale dei dati e di nuovo ogni volta che i dati cambiano [[7]](#Firebase_realtime_db_listener). Questa modalità di recupero dei dati è supportata in Java, Node.js e Python Admin SDK.

## Meteor

Meteor è un frame work di sviluppo javascript scritto in Node.js per lo sviluppo di applicazioni moderne multipiattaforma (Web, Android, iOS).

Si integra perfettamente al DBMS non relazionale MongoDB ereditando, di conseguenza, la sua espressività di interrogazione aggiungendo, inoltre, le query di aggiornamento tipiche dei sistemi database real-time [[8]](#Meteor).

Meteor utilizza il protocollo di dati distribuiti per propagare automaticamente le modifiche dei dati ai client senza richiedere allo sviluppatore di scrivere alcun codice per la sincronizzazione. Il protocollo dei dati distribuiti (DDP) è un protocollo client-server per l’interrogazione e l’aggiornamento di un database lato server e per la sincronizzazione di tali aggiornamenti tra i client [[9]](#meteor_ddp). Utilizza il modello di messaggistica Publish/Subscribe ed è stato creato per essere utilizzato dal frame work Meteor.

Il client meteor può essere implementato con qualsiasi framework JS frontend popolare: Vue, React, Angular.

## Parse

Simile a Meteor, infine, Parse è un frame work di sviluppo di applicazioni che utilizzano MongoDB come sistema dbms [[10]](#Parse1), è stato un frame work molto popolare e ha avuto una delle più grandi distribuzioni di MongoDB al mondo.

La società è stata acquistata da Facebook nel 2013 e chiusa nel gennaio 2017 divenendo open source.

Il videogioco Clash of Kings ha utilizzato Parse per l’implementazione delle notifiche push in tempo reale per gli utenti [[11]](#Parse2).

La maggior parte dei progetti che eseguono Parse provengono da applicazioni mobile (Android e iOS) e le tecnologie di frontend supportate comprendono Swift, Java, React Native, e altri [[12]](#Parse3).

Tecniche di auditing applicate ai dbms relazionali

L’unica tecnica applicata dai dbms relazionali che si avvicina concettualmente al design pattern *Publish/Subscribe* è la tecnica di auditing del database**.** Il database auditing implica, infatti, l' [osservazione di](https://en.wikipedia.org/wiki/Audit) un [database in](https://en.wikipedia.org/wiki/Database) modo da essere informati delle azioni svolte dagli [utenti](https://en.wikipedia.org/wiki/User_(computing)) [[13]](#Database_auditing).

[Gli amministratori](https://en.wikipedia.org/wiki/Database_administrators) e i consulenti di [database](https://en.wikipedia.org/wiki/Database_administrators) spesso impostano questi controlli per motivi di sicurezza al fine di vietare l’accesso ad alcune informazioni sul database a coloro che non hanno il permesso di lettura.

Attraverso le tecniche di database auditing sui dbms relazionali è quindi possibile monitorare e salvare le operazioni svolte su una tabella di un database [[14]](#Meccanismi_auditing) che diviene di conseguenza un’entità “attiva”.

Il concetto generale di database auditing riguarda il monitoraggio dell’utilizzo dei record di un database all’interno di un dbms relazionale [[15].](#SQLserver_audit)

Questa tecnica seppure si avvicini, non implementa completamente il design pattern Publish/Subscribe. Quando si controlla un database, infatti, le operazioni sui dati sono monitorate e registrate in un unico registro di controllo interno del dbms, chiamato audit trail [[16]](#why_you_need_auditTrail). Per questo motivo non è possibile scegliere di utilizzare un dbms relazionale come server di sistema in questa applicazione, perché non è garantita la notifica ai servizi client non rispettando, di conseguenza, il requisito principale del sistema da implementare.

Le tecniche di auditing hanno come obiettivo principale la sicurezza dei dati. In genere, l’auditing è utilizzato per:

* Assegnare a ogni utente delle “responsabilità” per le azioni eseguite in un particolare schema e tabella di un database.
* Impedire agli utenti del database di compiere azioni inappropriate considerando le “responsabilità” dell’utente stesso. L'implementazione degli audit trail aiuta a rendere il comportamento dell'utente più appropriato perché i suoi record utente possono essere ricondotti alla sua identità. Pertanto aiuta a prevenire fughe di dati importanti aziendali.
* Monitoraggio e raccolta di informazioni su attività specifiche del database. A volte gli audit trail possono essere utili per raccogliere informazioni statistiche.

### 

Le tecniche di auditing sono utilizzate soprattutto per prevenire la compromissione dei dati aziendali che possono essere eseguite dai dipendenti con accesso autorizzato.

L’auditing può essere soddisfatto implementando meccanismi di controllo nativi del dbms stesso oppure è possibile utilizzare software esterni dedicati. Occorre specificare, però, che l’audit nativo di ciascun dbms tende a causare un sovraccarico sul server del database.

I dati di audit nativi dei dbms, inoltre, non sono acquisiti nel formato richiesto dai revisori e dai team di sicurezza [[17]](#DataSunrise), pertanto, la soluzione migliore per soddisfare queste esigenze è l’integrazione di un software autonomo dedicato (es. DataSunrise, Imperva Data Activity Monitoring).

## RabbitMQ

RabbitMQ è sicuramente una diversa tecnologia che potrebbe essere utilizzata per implementare la gestione dello scambio dei messaggi tra le diverse entità del sistema [[18]](#RabbitMQ). Rappresenta, infatti, un [message-oriented middleware](https://it.wikipedia.org/wiki/Message-oriented_middleware" \o "Message-oriented middleware)  chiamato [broker di messaggistica](https://it.wikipedia.org/w/index.php?title=Broker_di_messaggistica&action=edit&redlink=1) che andrebbe a svolgere il ruolo di server del sistema implementato.

Un [broker di messaggi](https://en.wikipedia.org/wiki/Message_broker) è un programma intermedio che traduce un messaggio dal protocollo di messaggistica assunto dal mittente al protocollo di messaggistica del ricevitore. Rappresenta, inoltre, un modello architetturale per il routing dei messaggi implementando il pattern Publish/Subscribe. Il suo scopo principale è di recepire i messaggi in arrivo dalle applicazioni publisher ed eseguire alcune azioni su di essi. Ad esempio, il broker di messaggi può essere utilizzato per gestirne una coda da più ricevitori, fornendo memoria affidabile, garantendone, soprattutto, la consegna.

Nel tipico scenario dello scambio di messaggi tra le diverse entità coinvolte, RabbitMQ introduce, oltre la presenza del Publisher, del Consumer e della Queue, un nuovo elemento chiamato Exchange. Il Publisher definito da RabbitMQ non invia più il messaggio direttamente alla coda, ma passa dall’Exchange, il quale crea a sua volta la comunicazione con la coda stessa.

RabbitMQ offre la possibilità di utilizzare diversi tipi di Exchange in base al proprio contesto.

Si possono, tuttavia, mettere in evidenza tre principali categorie:

* Fanout
* Direct
* Topic

Ognuna di queste categorie definisce un diverso comportamento rispetto a come è indirizzato il messaggio dall’Exchange alle code. Utilizzando l’Exchange viene a determinarsi il sistema definito dal pattern Publish/Subscribe. Nella realtà, spesso, non sarà presente un solo consumatore ma migliaia, per questo motivo il messaggio pubblicato dal Publisher è inviato a tutte le code sottoscritte a uno specifico Exchange relativo al Publisher mittente.

## 

## Conclusioni sulle altre tecnologie utilizzabili

In conclusione, le tecnologie utilizzabili per l’implementazione del server di sistema sono tutti i software e applicativi in grado di implementare il design pattern *Publish/Subscribe*.

In questo progetto si è scelto sin dall’inizio di sviluppare il design del sistema indipendentemente dal tipo di tecnologia scelto per il server. Per questo motivo, se in futuro sarà necessario utilizzare un diverso sistema dbms o un’altra tecnologia, si potranno applicare poche e semplici modifiche di refactoring sulle classi di libreria implementate di *UtilityRethink* (vedi capitolo 4).

Facendo riferimento alle specifiche architetturali del sistema precedentemente discusse (vedi pag. 9), nel caso si scegliesse di utilizzare Firebase come server di sistema non sarebbe necessario utilizzare frame work aggiuntivi per la distribuzione delle notifiche perché offre questo tipo di “servizio”. Al contrario, gli altri sistemi analizzati, Parse e Meteor sono due diversi frame work che potrebbero essere utilizzati per la gestione delle notifiche appoggiandosi, però, al dbms non relazionale MongoDB. RabbitMQ, invece, è un sistema che garantisce la distribuzione delle notifiche ai vari servizi client ma dovrebbe essere affiancato da un sistema dbms per poter archiviare le stesse.

**Capitolo 3**

# Scelte architetturali per il caso d’uso aziendale

Nella fase iniziale, è stato ricercato il miglior software utilizzabile per la realizzazione del sistema. Per il contesto di utilizzo aziendale dell’applicativo, è stato deciso di affidare il ruolo di server al dbms RethinkDB [[22]](#RethinkDb). Il progetto di libreria *UtilityRethink*, presentato nel capitolo successivo, è stato implementato sull’ambiente di sviluppo Visual Studio ed è al momento disponibile solo la versione per applicativi *C#.*

Le funzionalità offerte da Git, infine, sono state utili alla creazione e mantenimento del repository di progetto online.

## Software utilizzato per il server

RethinkDb

Tra tutti i software elencati e descritti nel secondo capitolo della relazione, che potevano essere utilizzati per svolgere il ruolo di server del sistema, è stato scelto di utilizzare RethinkDB.

RethinkDB è un dbms non relazionale di tipo documentale [[23]](#RethinkDb_domande_frequenti) che appartiene alla classe di sistemi database real-time discussi nel paragrafo introduttivo.

Come nella maggior parte dei sistemi dbms non relazionali anche i dati presenti in una tabella di un database RethinkDB non seguono uno schema fisso predefinito. I documenti all’interno di una stessa tabella, infatti, possono avere campi diversi e, per questo motivo, è stato possibile mantenere tutte le notifiche di sistema in un’unica tabella (*Notifications*).

RethinkDB è il primo dbms non relazionale scalabile e open source costruito appositamente per il Web in tempo reale. Essendo un sistema database real-time, si basa su un’architettura fondamentalmente diversa dalla maggior parte dei dbms tradizionali (es MongoDB) esponendo una nuova modalità di accesso. Invece di eseguire il polling per le modifiche avvenute sui dati di un database, sovraccaricando il sistema, lo sviluppatore può richiedere a RethinkDb di inviare continuamente i risultati di query alle applicazioni client in tempo reale.

L’architettura push in tempo reale di RethinkDB è tipica dei sistemi database real-time, riduce i tempi e gli sforzi necessari allo sviluppo di applicazioni scalabili e real-time.

RethinkDB è, quindi, una buona scelta per il proprio progetto quando l’applicazione necessita di sapere dei cambiamenti dei dati in tempo reale. Per questa motivazione, quindi, è risultata essere una scelta ottimale per la realizzazione del server di sistema.

Altri casi d’uso in cui gli applicativi hanno beneficiato dell’architettura push in tempo reale offerta da RethinkDB sono:

* Web collaborativo e app mobile
* Streaming di applicazioni di analisi
* Giochi multiplayer
* Mercati in tempo reale
* Dispositivi collegati

Per il contesto aziendale, è stato scelto di utilizzare RethinkDB perché, rispetto a tutti gli altri software database real-time, forniva una migliore documentazione online

e garantiva lui stesso la funzionalità di distribuzione delle notifiche.

Analizzando le specifiche architetturali descritte nel capitolo 1 (vedi pag. 9), per questo motivo, il server di questo progetto è stato implementato utilizzando esclusivamente il dbms RethinkDB senza utilizzare un frame work aggiuntivo che gestisca il processo di notificazione (vedi capitolo 5).

Essendo, inoltre, un sistema dbms, utilizzando RethinkDB è stato possibile archiviare e salvare le notifiche inviate ai servizi client. In questo modo la libreria *UtilityRethink* implementata e discussa in seguito, garantisce all’utente anche la possibilità di poter controllare, in un secondo momento, le notifiche che sono state inviate perché sono salvate sul database RethinkDB. Ad esempio, la funzionalità di libreria *IQueryNotification* garantisce all’utente di richiedere tutte le notifiche avvenute in una specifica data.

RethinkDB è, inoltre un sistema diverso dalle API di sincronizzazione in tempo reale come FireBase, per tre diverse motivazioni [[24]](#RethinkDbvsAPI_sinc_in_temporeale):

* le API di sincronizzazione in tempo reale sono servizi cloud, mentre RethinkDB è un progetto open source. Sebbene RethinkDB sia disponibile anche nel cloud tramite Compose.io e Amazon AWS, può anche essere distribuito nelle proprie infrastrutture senza restrizioni;
* i sistemi come Firebase, sono limitati alla sincronizzazione dei documenti, mentre RethinkDB è un sistema dbms. Utilizzando RethinkDB è, quindi, anche possibile eseguire query arbitrarie tra cui operazioni di join tra tabelle, sottoquery, query geospaziali e query di aggregazione attraverso il linguaggio di interrogazione fornito chiamato ReQL. I servizi di sincronizzazione in tempo reale, invece, hanno capacità di interrogazione molto più limitate;

* le API di sincronizzazione in tempo reale sono progettate per essere accessibili direttamente dal browser, semplificando l’istallazione e l’esecuzione di applicativi di base, limitando, però, la flessibilità man mano che l’applicazione si “espande”. RethinkDB è progettato per essere accessibile da un server di applicazioni, in modo simile a un sistema dbms tradizionale. Ciò ha richiesto, per lo sviluppo del server di sistema, un codice di istallazione leggermente più complicato (vedi in seguito paragrafo Docker) ma ha consentito una maggiore flessibilità nel caso in cui, in futuro, il sistema diventi più “sofisticato”.

In generale, come per quasi tutti i dbms non relazionali, RethinkDB non è invece una buona scelta quando:

* si ha la necessità di rispettare le proprietà ACID ovvero atomicità, coerenza, isolamento e durabilità dei dati;
* è necessario rispettare lo schema di una tabella, ovvero tutti i record devono presentare gli stessi campi;
* si eseguono analisi ad alta intensità di calcolo. In questo caso occorre affidarsi ad un sistema Big Data come Hadoop o ad un archivio orientato alle colonne come Vertica;
* si ha la necessità di avere una forte disponibilità in scrittura dei dati. In alcuni casi RethinkDB, infatti, scambia la disponibilità in scrittura a favore della coerenza dei dati.

ReQL

Per interrogare un server RethinkDb, come già anticipato, viene utilizzato il linguaggio ReQl [[25]](#ReQL) .

Il ReQL è un linguaggio che permette di manipolare i documenti JSON di una tabella presente in un database RethinkDB in maniera efficace, riuscendo ad eseguire tutte le interrogazioni implementabili attraverso il linguaggio SQL classico. Permette, infatti, di eseguire operazioni di “join” tra tabelle, come mostrato in figura 3.1, e le funzioni di aggregazione.



Figura 3.1: esempio di query di join con linguaggio SQL e ReQL

SQL e RethinkDB condividono, inoltre, una terminologia molto simile. In figura 3.2 è presente una tabella dei termini e dei concetti dei due sistemi.

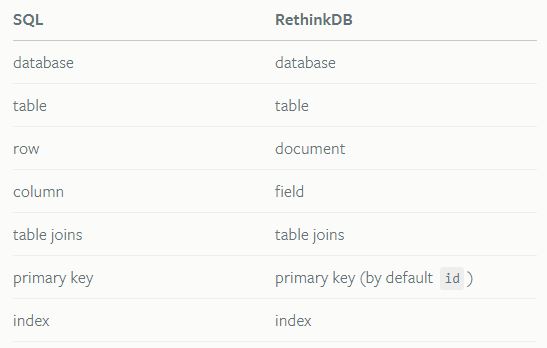


Figura 3.2: similitudine dei termini adottati da SQL e RethinkDB

Il linguaggio ReQL è diverso dagli altri linguaggi di query NoSQL perchè le interrogazioni al database vengono “costruite” attraverso chiamate di funzione nel linguaggio di programmazione scelto; nel caso della libreria di sistema *UtilityRethink*, in particolare, si è utilizzata l’API per applicativi in c# *RethinkDB/BChavez*.

Tutte le query ReQL, inoltre, sono concatenabili attraverso l’operatore “.” e, nonostante siano costruite sull’applicativo client, sono eseguite interamente sul server del database attraverso la chiamata del comando *run* a cui passare l’oggetto che rappresenta la connessione attiva al database. Nella libreria UtilityRethink la connessione al database è rappresentata dal campo *conn* di *ConnectionNodes*.

Docker (parlarne solo nel capitolo 5 nell’implementazione del server?)

Docker Desktop è un ulteriore software fondamentale per la realizzazione e gestione del server di sistema [[26]](#Docker) .

Docker Desktop è un’applicazione delle macchine MacOS e Windows per progettare e distribuire applicazioni e micro servizi containerizzati. Consente agli sviluppatori di utilizzare immagini e modelli certificati. I flussi di lavoro di progettazione sfruttano la piattaforma Docker Hub per estendere ogni ambiente di sviluppo a un repository sicuro per la creazione automatica rapida dello stesso, l’integrazione continua e la collaborazione sicura. Queste caratteristiche sono state sfruttate per l’implementazione del server che sarà descritta in dettaglio nel capitolo 5.

Un container è un’unità software che impacchetta il codice e tutte le sue dipendenze in modo che l’applicazione sia eseguita in modo affidabile da un ambiente di elaborazione a un altro. Un’immagine del container Docker rappresenta, invece, un pacchetto software leggero ed eseguibile che include il necessario per l’esecuzione dell’applicazione.

Le immagini dei container diventano contenitori in fase di esecuzione e, in particolare, nel caso dei Docker container, le immagini diventano contenitori quando vengono eseguite su Docker Engine. Il software containerizzato sarà, infine, eseguibile sempre allo stesso modo, indipendentemente dall’infrastruttura/sistema operativo [[27]](#docker_container) .

E’ stata utilizzata la versione Docker Desktop for windows perché per il contesto aziendale l’applicazione del progetto si esegue sul sistema operativo Windows[[28]](#docker_for_windows).

I comandi offerti da Docker Desktop, come sarà analizzato nello specifico in seguito, sono stati fondamentali per l’implementazione e mantenimento del server Rethinkdb containerizzato.

## Software utilizzati per l’implementazione della libreria di sistema Utility Rethink (parlarne del capitolo di UtilityRethink?)

Utilizzando lo store NuGet di Visual Studio è possibile scaricare diverse librerie di supporto ai propri progetti. Per l’implementazione della libreria di sistema UtilityRethink, sono quindi state scaricate e utilizzate le librerie *Bchavez/ RethinkDb.Driver*, *Reactive Extensions (Rx)* e *Simple Injector*.

*Bchavez/RethinkDb.Driver* è un driver che fornisce, agli applicativi scritti in linguaggio c#, un’api utile per l’interfacciamento al server Rethinkdb [[29]](#Bchavez_RethinkDb_driver). Pur non essendo un driver “ufficiale”, è molto simile a quello scritto in java, ed è stato fondamentale per lo sviluppo della libreria di sistema.

Per implementare il notificatore, oltre alla funzionalità di *changefeed* offerta dal driver *Bchavez/ RethinkDb.Driver,* è stata utilizzata anche la libreria *Reactive Extensions (Rx).* *Reactive Extensions (Rx)* è una libreria per lo sviluppo di programmi asincroni basati su eventi utilizzando il pattern observer tipico della programmazione a oggetti e operatori in stile LINQ [[30]](#Reactive_extensions).

*Simple Injector*, infine, è stato utilizzato, per il contesto aziendale, dagli applicativi sofware di frontend e beckend aziendali che utilizzeranno la libreria per gestire e richiamare le funzionalità dell’ applicativo stesso. Grazie ad esso è, infatti, possibile registrare un’istanza di una classe e fare in modo che quella restituita sia sempre la stessa al momento della registrazione [[31]](#simple_injector). Attraverso questo meccanismo offerto dalla libreria *Simple Injector* è stato possibile implementare automaticamente il Pattern Singleton sulla classe principale *UtilityRethink*.

## Metodologia di Progettazione (Togliere ?)

In seguito alla scelta del software da utilizzare per la realizzazione del sistema, è stato scelto Scrum [[32].](#Scrum) Scrum è una metodologia di progettazione agile che permette la divisione del progetto in intervalli di tempo di durata fissa generalmente da uno a quattro settimane chiamati Sprint.

Gli Sprint sono l’unità di base dello sviluppo in Scrum. Ognuno di essi è preceduto da una riunione di pianificazione in cui vengono identificati gli obiettivi e vengono stimati i tempi. Durante uno sprint non è permesso cambiare gli obiettivi, quindi le modifiche sono sospese fino alla successiva riunione di pianificazione, e potranno essere prese in considerazione nel successivo Sprint.

Durante lo sviluppo dell’applicazione, ogni attività svolta ha assunto i seguenti stati: “ToDo”, “Doing” e “Done”.

Scrum è una metodologia di progettazione che si utilizza quando i requisiti sono ben chiari e una volta iniziato il progetto non ci sono nuove attività che entrano nella lista “ToDo”. A tal proposito, per la gestione del progetto ho utilizzato Jira Software attraverso credenziali che mi sono state messe a disposizione dall’azienda. Quando iniziavo una nuova attività (Sprint) essa passava dallo stato “ToDo” al “Doing” per poi andare in “Done” quando veniva completato.

**Capitolo 4**

# Libreria UtilityRethink

Una delle due attività principali del progetto ha previsto l’implementazione di una libreria che garantisce l’interfacciamento al server di sistema agli applicativi aziendali.

Come tecnologia server è stato scelto il sistema dbms non relazionale real-time chiamato RethinkDB che sarà analizzato in seguito.

La libreria di utility sviluppata per fornire l’interfacciamento a RethinkDB, si chiama *UtilityRethink* e, come già introdotto nei capitoli precedenti, è stata implementata per garantire e ottenere il sistema di notifiche richiesto dall’azienda. E’, al momento, disponibile solamente una versione per applicativi scritti in linguaggio c# che sarà approfondita dettagliatamente in questo capitolo.

## Diagramma dei casi d’uso

Nello schema in figura 4.1 sono descritte le quattro principali funzionalità che la libreria offre ai servizi client che ne usufruiscono.

In particolare garantisce:

* la gestione del database sul server RethinkDB su cui sono archiviate le notifiche creando e eliminando tabelle e indici;
* l’inserimento di nuove notifiche relative ad un evento sul database RethinkDB;
* la ricezione di notifiche di interesse per le quali ci si è registrati;
* la ricerca delle notifiche presenti sul database RethinkDB;

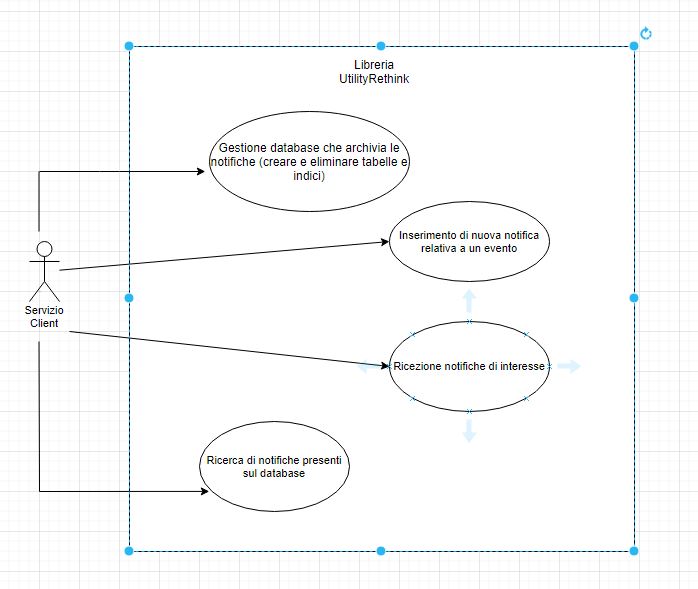


Figura 4.1

## 

## Diagramma delle attività e di sequenza

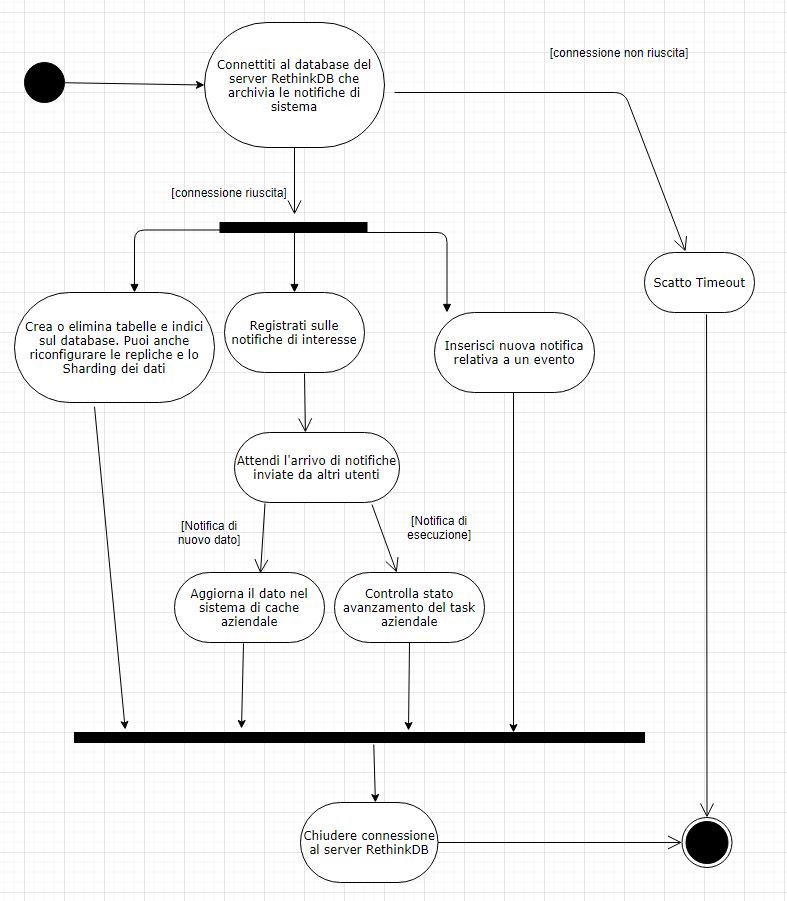


Figura 4.2

In figura 4.2 sono analizzati gli scenari possibili di utilizzo della libreria *UtilityRethink* da parte di un servizio client aziendale. L’utente che utilizza la libreria deve inizialmente connettersi al server RethinkDB per poter interagire con il database su cui sono archiviate le notifiche di sistema. Se, dopo un tempo impostabile dall’utente stesso, la connessione al server è rifiutata viene fatto scattare il timeout.

Successivamente alla connessione al database RethinkDB, l’utente può:

* Registrarsi su alcune notifiche di interesse;
* Eseguire operazioni di management del database: riconfigurare i parametri di Sharding e Replication dei dati e creare o eliminare tabelle e indici;
* Inserire sul database una nuova notifica relativa ad un evento;

Il servizio client, in seguito alla registrazione su alcune notifiche di interesse, può inserire nuove notifiche, eseguire operazioni gestionali sul database e rimanere in attesa di ricevere notifiche di interesse.

Quando altri utenti inseriscono sul database una nuova notifica, come mostrato nel diagramma di sequenza in figura 4.3, RethinkDB controlla tutti i servizi client che si sono registrati in precedenza sulle notifiche aventi lo stesso argomento. Se il *servizio client 1* si era interessato all’argomento di quella notifica sarà informato da RethinkDB della nuova inserita nel database.

Quando un servizio client viene “notificato” è libero di decidere come agire. Come descritto nel diagramma in figura 4.2, i servizi client aziendali utilizzeranno questo sistema di notificazione per aggiornare i dati nel sistema di cache aziendale se la notifica arrivata è di tipo *nuovo dato* mentre controlleranno lo stato di avanzamento di un task/processo aziendale se la notifica arrivata è di tipo *esecuzione*.

Le notifiche e il sistema di notificazione saranno, in seguito, trattate nel dettaglio.

## C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\DiagDiSequenza.JPG

Figura 4.3

## Base Dati

Per la progettazione del database è stato necessario considerare l’utilizzo di un'unica tabella, perché il sistema aveva come unico requisito quello di mantenere le notifiche relative a eventi generati da applicativi aziendali. Questa tabella è stata chiamata *Notifications* e dovrà raccogliere e mantenere ogni tipo di notifica di sistema. In tutti e tre i server RethinkDB che sono stati implementati in questo progetto (uno, due, cinque nodi) è stata inserita la tabella *Notifications*.

Notifiche



Figura 4.4: notifiche sulla tabella di sistema *Notifications.*

RethinkDB, essendo un dbms di tipo non relazionale, accetta la diversità dei campi tra documenti di una stessa tabella. Diversamente dai dbms relazionali, questi sistemi non seguono uno schema fisso per le proprie tabelle. E’ stata, quindi, sfruttata questa caratteristica, tipica dei dbms non relazionali, per raccogliere due diversi tipi di notifiche all’interno della stessa tabella *Notifications* come mostrato in figura 4.4.

L’unico vincolo richiesto è che, in futuro, qualsiasi nuovo tipo di notifica che si inserirà su *Notifications,* dovrà obbligatoriamente avere i campi della classe astratta *Notification* di *UtilityRethink* mostrata in figura 4.5.

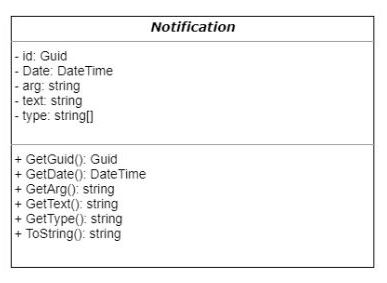


Figura 4.5: classe base delle notifiche *Notification*.

Ogni classe c# che rappresenta un tipo di notifica deve *ereditare* dalla classe base *Notification* sopra elencata in figura 4.5.

Per il contesto aziendale, UtilityRethink offre due tipi di notifiche chiamate *NotificationNewData* e *NotificationExec* che, causa il vincolo impostato, ereditano tutti i campi e le proprietà della classe *Notification.*

Se in futuro il cliente della libreria avesse la necessità di considerare nuovi tipi di notifiche, per mantenerle sulla tabella *Notifications* nei server RethinkDB implementati, non sarà un problema. Le funzionalità gestionali offerte lavorano, infatti, con qualsiasi tipo di “classe notifica” che *eredita* da *Notification*. La classe c# che “mapperà” il nuovo tipo di notifica dovrà, quindi, come unico requisito ereditare dalla classe base di libreria delle notifiche *Notification*.

Per implementare questa soluzione sono state utilizzate due proprietà fondamentali della programmazione a oggetti: la Reflection e i generici.

Notifiche di nuovo dato

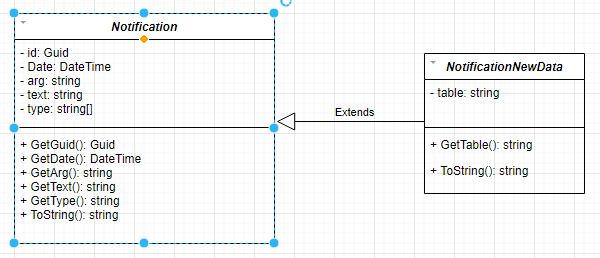


Figura 4.6: notifiche di nuovo dato

Le *notifiche di nuovo dato* sono chiamate *NotificationNewData* dalla libreria *UtilityRethink* e, ereditando da *Notification*, come mostrato in figura 4.6, hanno tutti i campi base delle notifiche. Inoltre hanno un campo “*Table*” che serve a identificare una tabella del database reale aziendale su cui è avvenuta una modifica. Queste notifiche raccolgono su quale tabella è avvenuta un’operazione di *insert*, *update* o *delete* di un dato.

Oltre a tracciare in real-time le operazioni svolte sulle tabelle del database reale grazie all’utilizzo delle notifiche di nuovo dato, i servizi software aziendali che utilizzano la libreria saranno in grado di aggiornare i dati presenti sul sistema di caching aziendale.

Il sistema di caching è uno strato software che s’interpone tra i servizi client e il sistema di archiviazione dei dati ed è, inoltre, un livello di storage ad alta velocità che memorizza un sottoinsieme di informazioni solitamente molto richieste per rispondere più rapidamente.

Questo sistema viene, quindi, molto utilizzato dall’azienda Energy Software per chiedere i dati in maniera più performante rispetto a interrogare direttamente il vero sistema di archiviazione adottato. Sarebbe molto dispendioso, in termini di performance, se per ottenere i dati i servizi client aziendali interrogassero il database reale.

Queste notifiche permetteranno, quindi, di fornire informazioni costantemente aggiornate agli utenti delle applicazioni web aziendali.

Notifiche di esecuzione

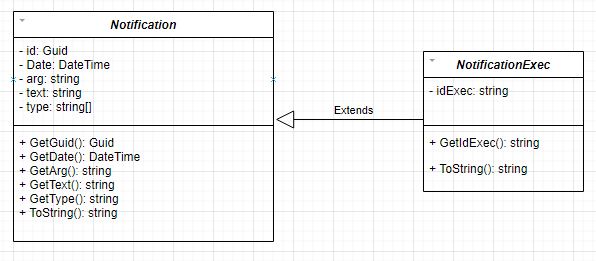


Figura 4.7: notifiche di esecuzione

Le *notifiche di esecuzione* sono invece “mappate” dalla libreria *UtilityRethink* come *NotificationExec*. Esse serviranno ad avvisare i servizi software aziendali interessati allo stato di avanzamento di un Task/processo.

Come le *notifiche di nuovo dato*, *le notifiche di esecuzione* ereditano tutti i campi e le funzionalità base dalla classe di libreria *Notification* ed hanno, in aggiunta, un campo *idExec* che rappresenta l’*id* del Task aziendale.

### 

## Diagrammi delle classi

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\StrutturaGeneraleUml.jpg

Figura 4.8: diagramma delle classi della libreria *UtilityRethink.*

### UtilityRethink

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\IUtilityRethink.JPG

Figura 4.9: UtilityRethink

*UtilityRethink* è la classe fondamentale della libreria che, attraverso i suoi sotto-componenti forniti, permette all’utente di interagire con un database presente su un server RethinkDB. I sotto-componenti forniti, come mostrato in figura 4.9, sono il manager del database chiamato *DbManager* e *INotificationsManager* ovvero il manager della tabella di sistema *Notifications*. Se in futuro si avrà la necessità di utilizzare nuove tabelle considerate dalla libreria “di sistema”, sarà sufficiente aggiungere alla classe principale *UtilityRethink* un riferimento al suo nuovo “manager” nominandolo seguendo il formato “INomeTabellaManager”.

Ogni volta che la classe principale di libreria *UtilityRethink* è istanziata, viene creata una nuova connessione ai nodi del server RethinkDB in ascolto sugli indirizzi specificati. Come spiegato nell’esempio in figura 4.10, l’applicativo client viene fatto “connettere” al database specificato. Il server su cui si connetterà l’applicativo, è formato da cinque nodi RethinkDB in ascolto, mentre il database di riferimento si chiama “test”. Se sul server RethinkDB non esiste quel database, ne viene creato uno nuovo con quel nome e sono inserite tutte le tabelle considerate dalla libreria di “sistema”. Come già specificato precedentemente, sarà inserita solamente la tabella *Notifications* perché il contesto aziendale richiedeva un’unica “tabella di sistema”.

Lo stesso oggetto che rappresenta la connessione al database richiesto, generato in seguito all’istanziamento di *UtilityRethink*, viene infine “passato” a tutti i sotto-componenti principali di libreria (*DbManager* e *INotificationsManager)* per poter essere utilizzato come tramite con il server RethinkDB.

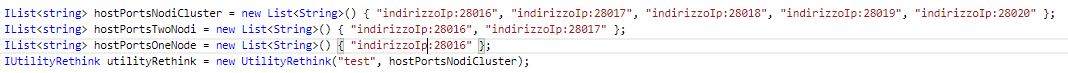


Figura 4.10

#### 

### Simple Injector

Nel codice degli applicativi di frontend e backend aziendali serviva la possibilità di richiamare la stessa istanza della libreria *UtilityRethink* in diversi punti del codice ed era, inoltre, necessario fornire un punto di accesso globale a tale istanza.

A tal proposito, sono stati approfonditi il design pattern *Singleton* e il *Factory Method*, tipici della programmazione a oggetti, per utilizzare correttamente la libreria *Simple Injector*. Attraverso la classe *Container*, di *Simple Injector*, è possibile registrare un’istanza di una classe come se fosse un “oggetto singleton”. Successivamente alla registrazione, la stessa istanza può essere richiamata in diversi punti del codice come mostrato in Figura 4.11. Ho sfruttato, quindi, questa caratteristica della libreria *Simple Injector* per simulare il pattern *Singleton* sulla classe principale della libreria “UtilityRethink.cs”.

# 

Figura 4.11

### DbManager

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\DbManager.JPG

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\DbManagerCodice.JPG

Figura 4.12

Il *DbManager* permette all’utente di libreria di gestire il database specificato inizialmente all’istanziamento di *UtilityRethink*. Tra le funzionalità offerte troviamo la possibilità di creare e eliminare tabelle e indici come indicato in figura 4.12.

Il *DbManager* offre, infine, anche la possibilità di riconfigurazione delle repliche e delle shard dei dati di una tabella del database.

Attenzione: Non è possibile eliminare le tabelle considerate di sistema dalla libreria presenti sul database (es. *Notifications*).

### NotificationsManager

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\NotificationManager.JPG

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\NotificationsManager.JPG

Figura 4.13

Il *NotificationsManager* è il componente della libreria utile alla gestione della tabella di sistema *Notifications*. Come sottolineato precedentemente, se in futuro si dovesse avere la necessità di mantenere anche altre “tabelle di sistema” oltre a *Notifications* , si potrà semplicemente aggiungere il suo nuovo manager per poter essere gestito dalla classe principale di libreria *UtilityRethink*.

*NotificationsManager* può restituire all’utente di libreria, come mostrato in figura 4.13, due sotto componenti fondamentali alla gestione delle notifiche del sistema: *IQueryNotification* e *IRXNotifier*.

#### 

### IQueryNotification

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\QueryNotifications.JPG

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\QueryNotificationCodice.JPG

Figura 4.14

Il *QueryNotification* è una funzione specifica offerta da *NotificationsManager* che garantisce, all’utente di libreria, la possibilità di interrogare la tabella di sistema *Notifications* ed, inoltre, offre le funzionalità di inserimento e cancellazione di notifiche. Nel caso della cancellazione occorre specificare l’“id”della notifica da eliminare e, permette, infine, di richiedere notifiche con determinate caratteristiche come mostrato in figura 4.14.

*IQueryNotification* gestisce qualsiasi tipo di notifica presente sulla tabella *Notifications*. In figura 4.15 sono illustrati due esempi in cui si richiedono le notifiche di esecuzione e le notifiche di nuovo dato appartenenti ad una certa data. Come precisato precedentemente nel capitolo delle notifiche, questa soluzione è stata implementata utilizzando le proprietà dei generici e della Reflection tipiche della programmazione a oggetti.

C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\GetNotificationsNewData.JPG



Figura 4.15: QueryNotification per le notifiche di nuovo dato e di esecuzione

### IRXNotifier

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\RXNotifier.JPG

Figura 4.16

*IRXNotifier* è la classe di libreria che svolge il fondamentale ruolo di notificatore degli eventi del sistema.

La soluzione è stata implementata sfruttando una specifica funzionalità offerta dal dbms non relazionale RethinkDB. Questa caratteristica, chiamata *changefeeds*, è il motivo per cui RethinkDB può essere considerato un database real-time (classe di sistemi discussi nel paragrafo introduttivo) perché consente ai client di ricevere in tempo reale le modifiche avvenute su una tabella, su un singolo documento o persino i risultati di una query specifica. Il notificatore, quindi, è stato implementato attraverso la combinazione della funzionalità *changefeeds* offerta dal driver RethinkDB per applicativi in c# chiamato *bchavez/RethinkDb.Driver* e della libreria *Reactive Extension (Rx)* [[19]](#Consuming_changefeed).

*IRXNotifier* è un notificatore che, come il *QueryNotification*, offre le proprie funzionalità per la tabella di sistema *Notifications* permettendo agli applicativi client di rimanere in ascolto degli eventi che la riguardano.

Garantisce, inoltre, di intercettare eventi riguardanti un sottoinsieme specifico di notifiche con un certo valore di argomento (campo *arg* delle notifiche). Gli eventi comprendono l’inserimento, la cancellazione e la modifica di una notifica che abbia come argomento uno tra quelli per cui l’applicativo client si è interessato registrandosi. In seguito alla registrazione sulle notifiche di proprio interesse viene restituito agli applicativi client della libreria un *NotificationSubscription* che raccoglie un *observable* (oggetto che permette di intercettare gli eventi) con un suo id.

Ora, passaggio fondamentale, i clienti della libreria devono sottoscrivere ciascun oggetto *observable* a dei metodi, implementandoli in base alle proprie esigenze. Questi metodi a cui si devono sottoscrivere gli oggetti *observable* forniti dalla libreria sono: *OnNext*, *OnError* e *OnComplete*.

Ad esempio, come mostrato in figura 4.17, in caso di intercettazione corretta di un evento sarà eseguito il metodo *onNext*, in caso di errore il metodo *OnError* e al termine dell’ascolto (se verrà chiamato il metodo *StopListening* sull’*observable*) il metodo *OnComplete* [[20]](#notifier_es). Attraverso il metodo *OnNext*, in particolare, ogni applicazione client potrà gestire, in base al proprio contesto, la notifica arrivata utilizzando l’oggetto che in figura 4.17 è chiamato *x*.

Nel contesto aziendale, ad esempio, potrebbe essere stata inserita una nuova notifica sulla tabella *Notification*. Il sistema avviserà l’applicativo client della modifica avvenuta su una tabella del database reale aziendale. In questo modo ricevendo la notifica dell’aggiornamento (*notifica di nuovo dato*) si potrà procedere ad aggiornare il dato anche nel sistema di caching aziendale per garantire alle interfacce web di avere informazioni sempre aggiornate.

Esempio di utilizzo notificatore sulle notifiche di esecuzione:

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\ObservableNotificationExec.JPG

Figura 4.17: esempio di utilizzo notificatore sulle notifiche di esecuzione con argomento “A” e “B”

Come il *QueryNotification*, anche *IRXNotifier* offre le proprie funzionalità ad ogni tipo di notifica presente sulla tabella di sistema *Notification*.

In figura 4.17 l’applicativo rimane in ascolto per le notifiche di esecuzione che hanno come argomenti i valori “A” e “B”.

Quando le classi c# che rappresentano le notifiche ereditano da quella base *Notification* (come ad es. le *notifiche di nuovo dato*), è possibile ottenere un loro oggetto notificatore sostituendo dentro le “<>” al posto di “NotificationExec” con il loro nome della classe.

Anche per *IRXNotifier*, questa soluzione è stata implementata utilizzando le proprietà dei generici e della Reflection tipiche della programmazione a oggetti.

#### Esempio di OnNext, OnError e OnComplete

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\OnNextOnErrorOnCompleteCodice.JPG

Figura 4.18

### Connection

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\Connection.JPG

Figura 4.19

Il *ConnectionNodes* è la classe che rappresenta la connessione al server RethinkDB. Viene istanziata, inizialmente, dalla classe principale *UtilityRethink* per poter essere utilizzata da tutti i componenti della libreria come *IQueryNotifications* e *IRXNotifier*. I sotto-componenti di libreria, infatti, utilizzano per interfacciarsi al server la stessa istanza di *ConnectionNodes*.

*ConnectionNodes*, come mostrato in figura 4.19, memorizza tutti i nodi del server RethinkDB con cui l’utente di libreria interagisce attraverso le *DbOptions*. In questa applicazione, in particolare, possono essere 1, 2 o 5, in base al server messo in esecuzione. I dettagli del server RethinkDB implementato saranno trattati in seguito.

Il campo *conn* di *ConnectionNodes* rappresenta il vero oggetto di connessione al database c viene istanziato solamente, in seguito, alla prima interazione col server RethinkDB, ovvero alla prima esecuzione del metodo *GetConnection*.

Alle successive chiamate di *GetConnection*, dei vari componenti di libreria, non viene più istanziato ma viene sempre restituito lo stesso oggetto. Inizializzare più volte l’oggetto di connessione al server RethinkDB *conn* risulterebbe un’operazione pesante. *Conn* per connettersi ai nodi del server RethinkDB esegue un’algoritmo di RoundRobin, implementato dalla libreria *Bchavez/Rethinkdb*, che sceglie a run-time di interagire con un nodo disponibile al momento.

*ConnectionNodes* gestisce, infine, anche i tentativi di connessione su indirizzi Ip non raggiungibili [[21]](#connessione_roundRobin). Se, dopo 20 secondi, non si riesce a connettere al server viene fatto scattare il Timeout per la mancata connessione.

**Capitolo 5**

# Implementazione Server

Per riuscire a implementare il server RethinkDB di sistema, come anticipato nel capitolo precedente, è stata utilizzata la tecnologia offerta dai container Docker.

Attraverso Docker-hub, sito web su cui è possibile cercare e scaricare immagini docker di applicativi, è stata scelta di utilizzare l’immagine ufficiale *rethinkdb* mantenuta dalla omonima società RethinkDb [[33]](#immagine_rethinkdb). In seguito al download dell’immagine, scrivendo sul terminale i comandi di Docker-compose, offerti dall’applicativo Docker-Desktop, è stato possibile implementare tre diversi server RethinkDB.

Server a singolo nodo

Il server RethinkDB a singolo nodo, come rappresentato in figura 5.1, è un server che è costruito utilizzando un solo container Docker RethinkDB.

L’unico nodo disponibile di questo server, rimane in ascolto sulla porta 8081 e sarà raggiungibile da tutti gli applicativi che utilizzeranno la libreria di sistema UtilityRethink implementata.

Server1NodoTerminale

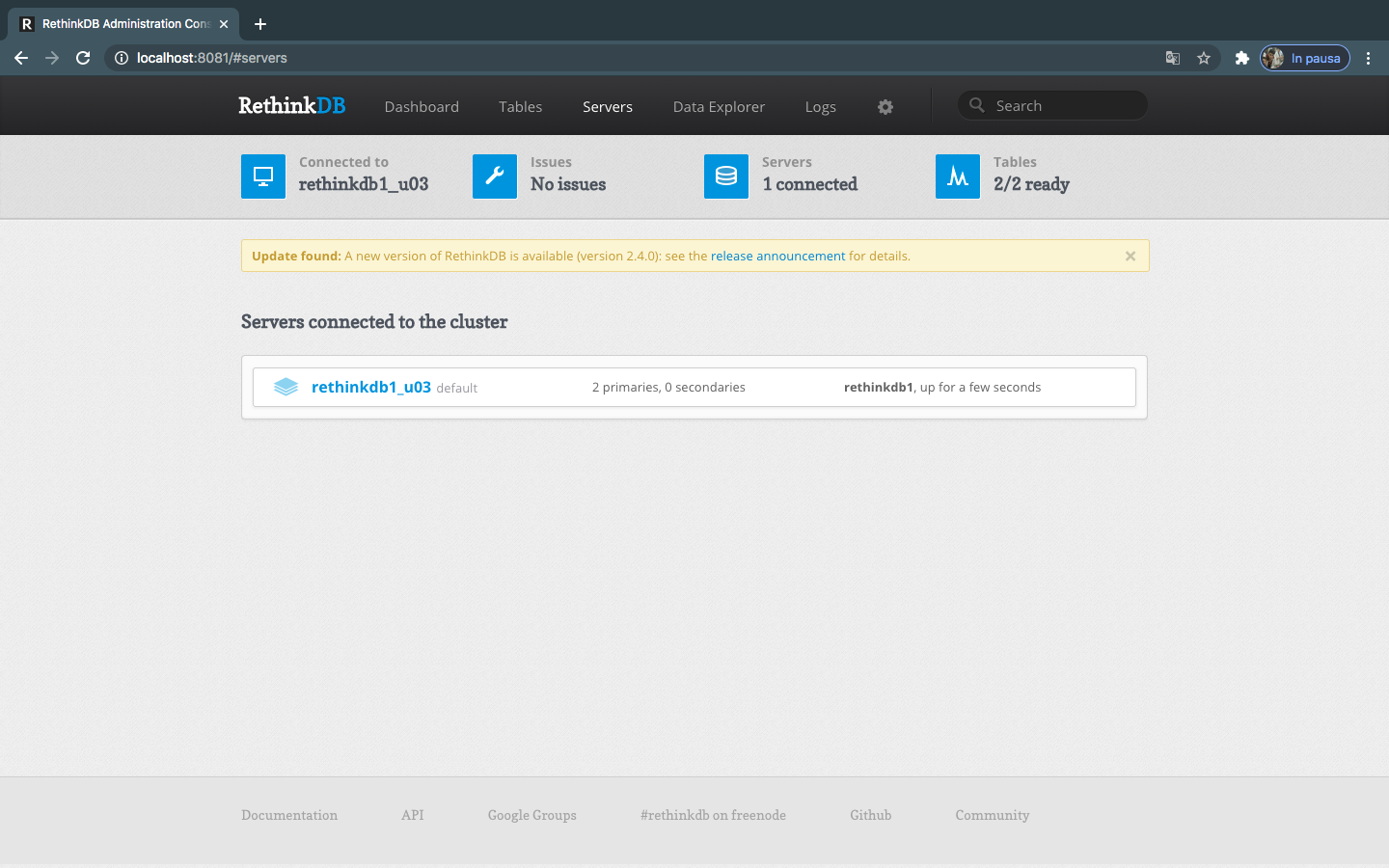


Figura 5.1

Server a due nodi

Il server a due nodi, invece, come mostrato in figura 5.2, è implementato attraverso due container Docker.

Utilizzando il sistema di tagging tipico di RethinkDB, che sarà trattato in seguito, è stato possibile implementare uno script bash, “*start-rethinkdb-cluster.sh*”, che, attraverso alcuni specifici comandi, ha garantito l’unione e la collaborazione dei due container Docker in un unico server RethinkDB.

Questo server, essendo formato da due nodi, è raggiungibile sul proprio indirizzo locale sulla porte 8081 e 8082.

Server2NodiTerminale

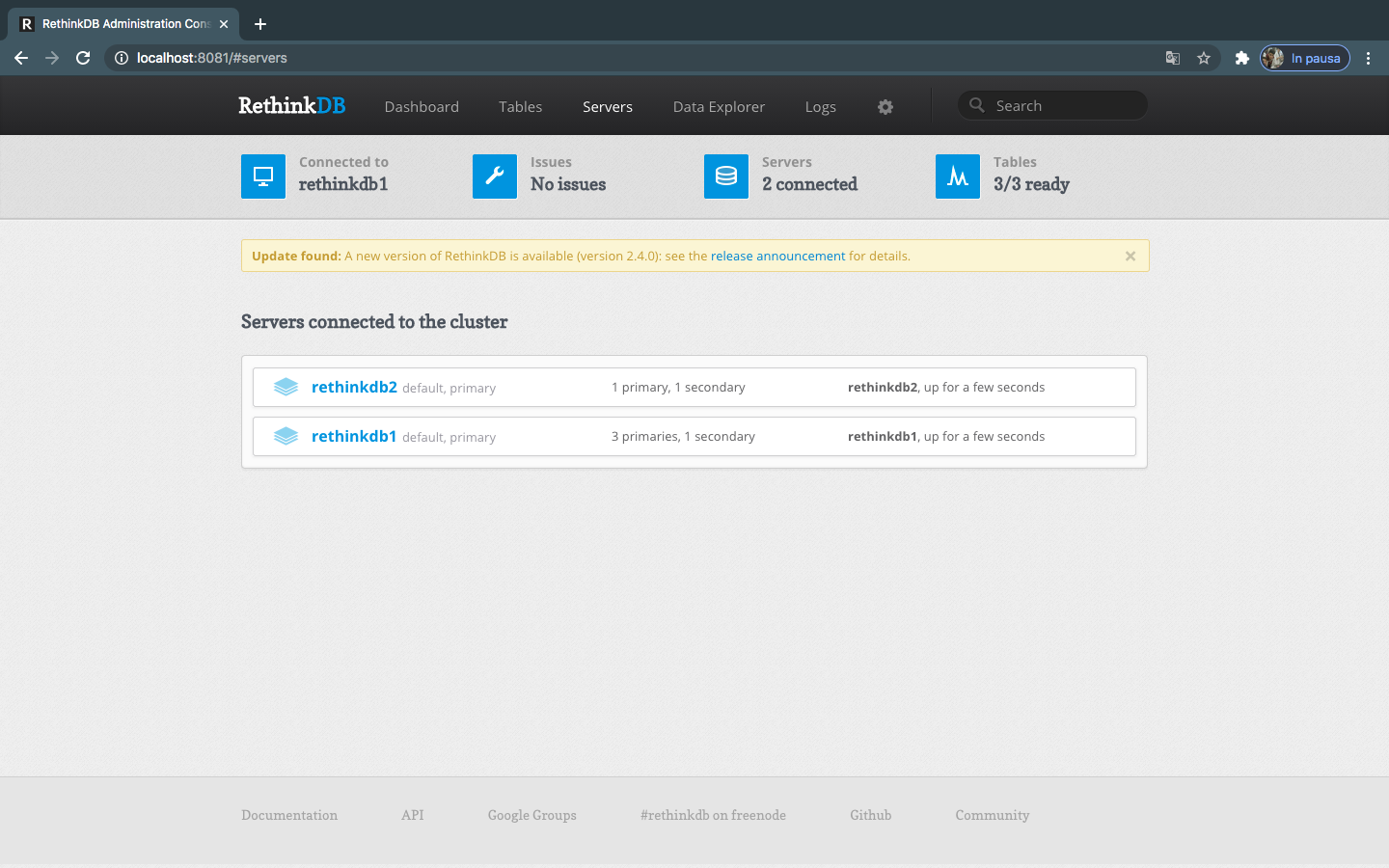


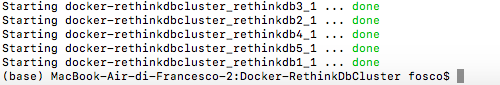
Figura 5.2

Server a cinque nodi

Il server a cinque nodi, infine come descritto in figura 5.3, è stato sviluppato attraverso cinque container Docker.

Utilizzando lo script bash “*start-rethinkdb-cluster.sh*”, come per il server a due nodi, è stato eseguito il join dei cinque nodi in un unico server RethinkDB. Questo server, essendo formato da cinque nodi RethinkDB, sarà raggiungibile sul proprio indirizzo Ip locale sulle porte 8081, 8082, *8083*, 8084 e 8085.

Saranno spiegati, in seguito, i comandi per costruire, eseguire e fermare i server RethinkDB implementati.



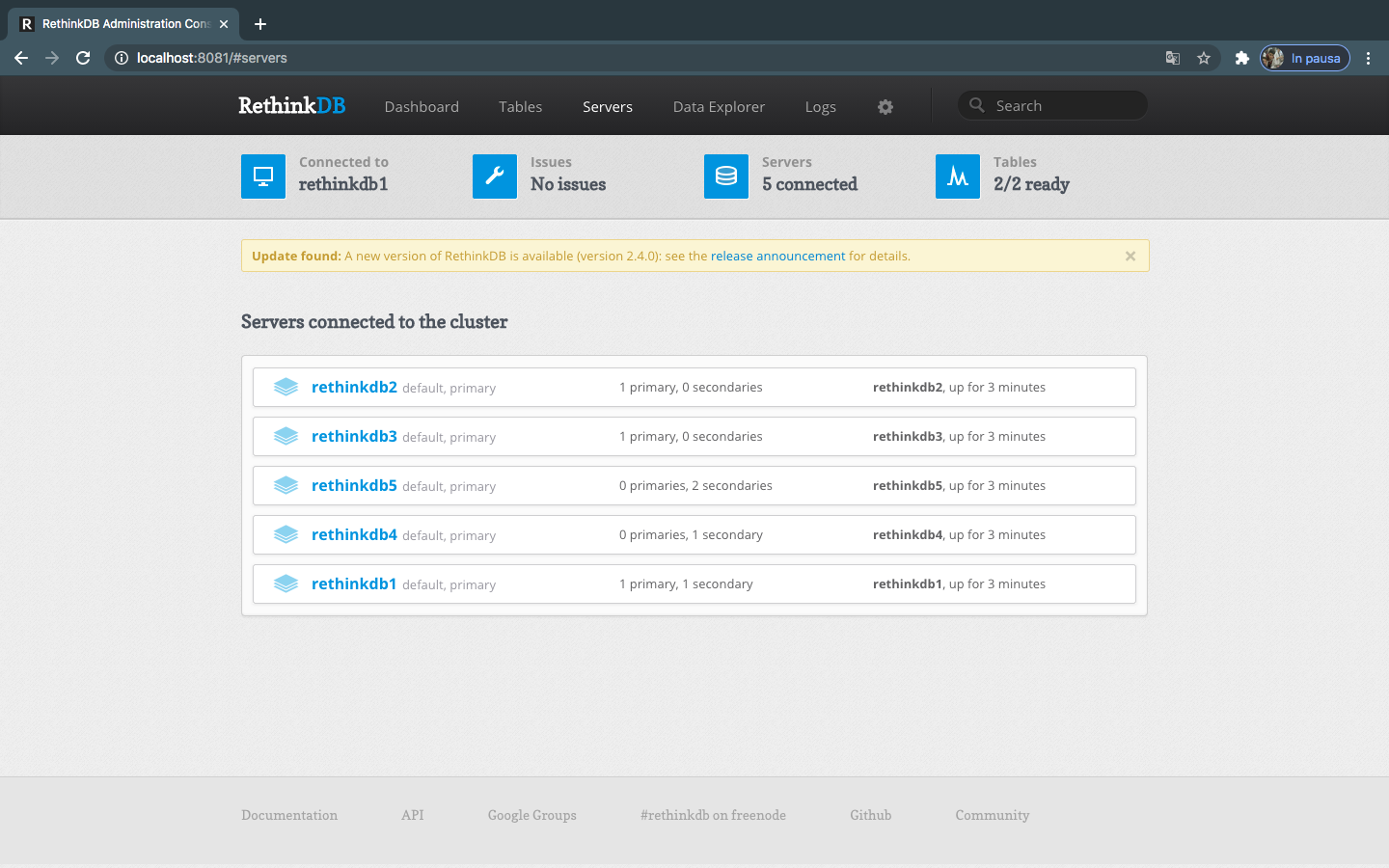


Figura 5.3

Conclusioni sui tre server implementati

Ogni nodo dei tre server RethinkDB, quindi, rappresenta di fatto un container. Nei due casi in cui il server è formato da più nodi, è stato utilizzato un particolare “sistema di tagging” tipico di RethinkDB[[37]](#RethinkDb_tag). Questi *tag* sono definiti dall’utente e servono a identificare i nodi. A ogni nodo in un cluster RethinkDB, come indicato in figura 5.4 è possibile assegnare zero o più *tag*.



Figura 5.4

Nei due server a più nodi, il sistema di tagging è stato fondamentale per identificare i nodi/container RethinkDB. In questo modo, nello script bash *start-rethinkdb-cluster.sh* sono eseguitii comandi necessari per il join dei nodi in un unico server [[34]](#RethinkDb_join_cluster).

Grazie a questo particolare script, i nodi di uno stesso server RethinkDB collaborano tra di loro per riuscire a rispondere alle richieste in arrivo dai servizi client e per sfruttare, in maniera efficace, le politiche di Replication e di Sharding.

In questa applicazione del sistema, bisogna specificare che il server risiede, al momento, sulla macchina locale, perché sono stati utilizzati i comandi di Docker-compose [[35]](#Docker_compose) . In un contesto più articolato, è possibile utilizzare altre tecnologie per distribuire i diversi nodi del server su più macchine fisiche. Questa potrebbe essere una probabile modifica futura per l’applicativo e Docker Swarm o Kubernetes sarebbero, ad esempio, alcune tecnologie su cui fare riferimento.

## Comandi per la gestione del server di sistema

Per utilizzare uno dei tre server RethinkDB implementati occorre seguire la stessa procedura in tutti i casi.

Aprire il terminale e dirigersi sulla cartella del progetto corrispondente all’opzione del server che si vuole scegliere (uno, due o cinque nodi). Digitare, in seguito, sempre sul terminale, “*docker-compose -f docker-compose.yml build*” che, come mostrato in figura 5.5, costruirà l’immagine docker del server sul proprio computer. Questo comando deve essere eseguito solamente al primo utilizzo del tipo di server scelto, perché l’immagine costruita viene salvata permanentemente sul proprio dispositivo.



Figura 5.5: build dell’immagine del server RethinkDB a due nodi

Successivamente, è possibile controllare il numero di immagini salvate sul proprio dispositivo, controllandone anche lo stato (colonna “*status*”), attraverso il comando “*docker ps –a*”.

In seguito è possibile mettere in esecuzione il server RethinkDB scelto. Sempre attraverso il terminale, scrivere il comando “*docker-compose -f docker-compose.yml up -d*” per eseguire il server. Il server RethinkDB, come mostrato in figura 5.6, è ora online e i suoi nodi sono in attesa di rispondere alle richieste delle applicazioni client.

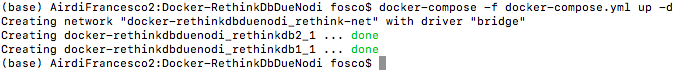


Figura 5.6: up del server RethinkDB a due nodi

Infine, come mostrato in figura 5.8, è possibile “stoppare” il server RethinkDb digitando, sempre sul terminale, il comando “*docker-compose -f docker-compose.yml stop*”.

E’ importante sottolineare che lo stato dell’immagine docker del server RethinkDB in seguito allo stop viene salvato.

Se sono inseriti dei nuovi dati sul database e, in seguito, il server è fermato, al prossimo comando di “up”, il server mantiene i dati che sono stati caricati in precedenza.

C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\fotoserver\Docker-stop.png

Figura 5.7: stop del server RethinkDB a due nodi.

## Sharding e Replication

Durante lo sviluppo di un cluster di un dbms non relazionale, occorre considerare le politiche di Sharding e Replication sui dati.

Lo Sharding serve a distribuire i dati al meglio sul Cluster. I dati vengono inizialmente suddivisi in piccole parti, dette shards, per essere successivamente distribuite in modo omogeneo nei diversi nodi del server. Una buona strategia di sharding è fondamentale per ottimizzare le performance. Le diverse partizioni dei dati, in seguito ad una richiesta, potranno essere lette in parallelo dai diversi nodi.

La Replication, invece, serve alla creazione e gestione di copie dei dati sui nodi del Cluster. E’ una tecnica che, nel caso di “node failure”, previene la perdita dei dati, perchè gli stessi sono stati precedentemente salvati anche su un secondo nodo. Garantisce, inoltre, la lettura dei dati in “parallelo” da diversi utenti perchè sono presenti repliche su più nodi del Cluster.

Utilizzando le tecniche di replicazione è possibile che la lettura di un dato avvenga da un nodo del server vicino geograficamente a dove è stato richiesto. Se esistesse una sola copia di un dato ci sarebbe alta probabilità che lo stesso risieda su un nodo particolarmente lontano rispetto a dove è stato richiesto. Ci sarebbero, di conseguenza, alti costi di rete.

E’ possibile gestire le politiche di sharding e replication su un cluster RethinkDB tramite interfaccia web o attraverso comandi ReQL [[36]](#rethinkdb_Sharding_replication) .

### Gestione con Interfaccia Web

Le impostazioni del server RethinkDB possono essere controllate facilmente dalla console di amministrazione web.  Quando si utilizza l'interfaccia utente Web, per gestire la Replication e lo Sharding dei dati, è sufficiente specificare il numero di frammenti desiderati e il numero di repliche dei dati richiesti. Sarà poi RethinkDB stesso, in base alle proprie disponibilità, a determinare i suoi migliori punti di divisione per mantenere i frammenti bilanciati.

Per replicare e dividere i propri dati:

Dirigersi sulla “vista” della tabella ( *Tabelle* → *nome tabella* ).

Fare clic sul pulsante *Reconfigure* mostrato in figura 5.8 .

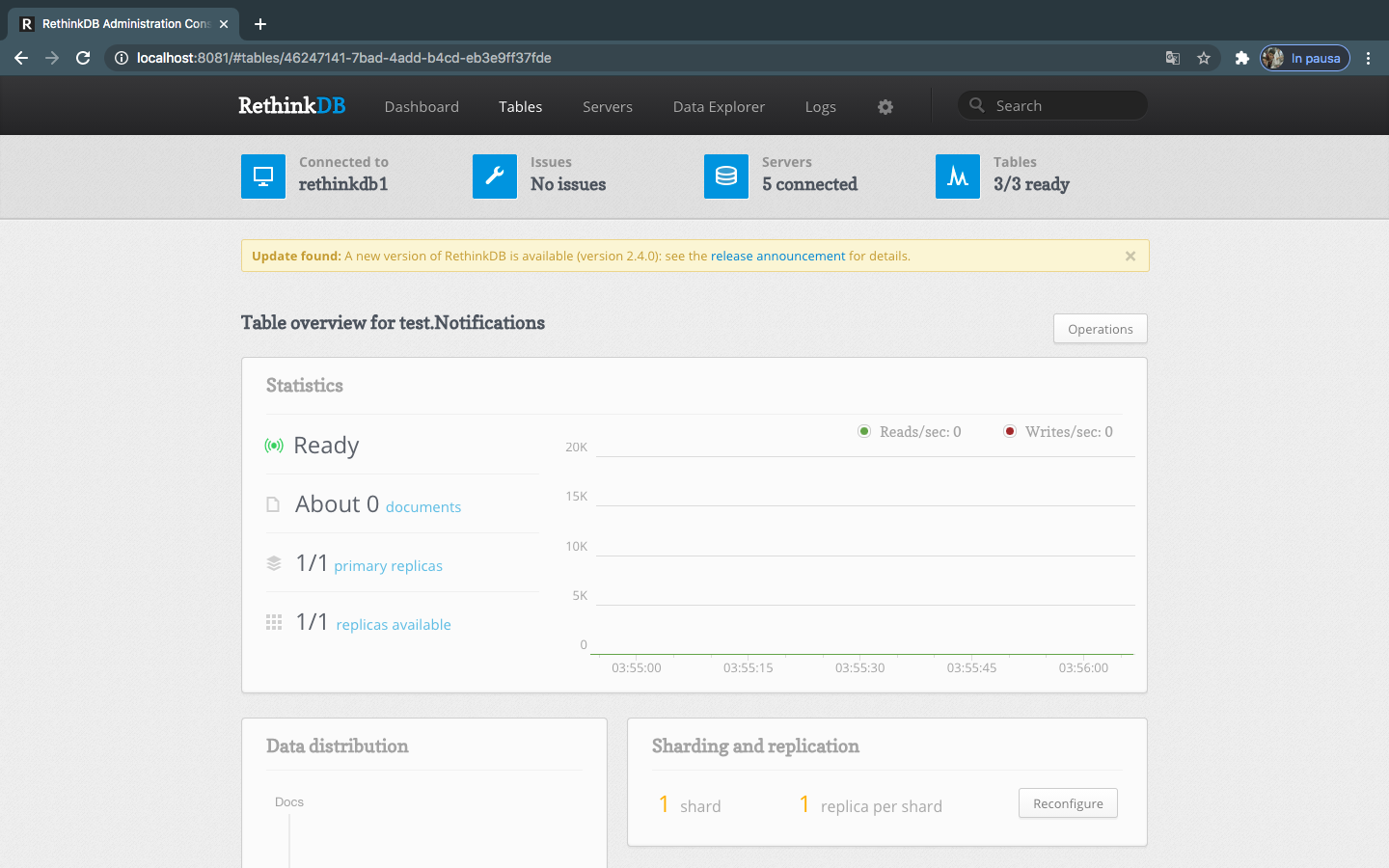


Figura 5.8: riconfigurazione dei dati della tabella di sistema Notifications

A questo punto, come mostrato in figura 5.9, è possibile impostare il numero di frammenti e repliche che si desiderano per i propri dati. Nell’esempio mostrato in figura, in particolare, si analizza il caso in cui si vogliano riconfigurare i parametri di Replication e Sharding sui dati della tabella di sistema *Notifications* presenti sul server a cinque nodi che

è stato implementato. Come indicato in figura 5.9, il limite massimo del numero di repliche e shard dei dati che è possibile impostare è cinque, perché il server, in questo caso, è formato da cinque nodi.

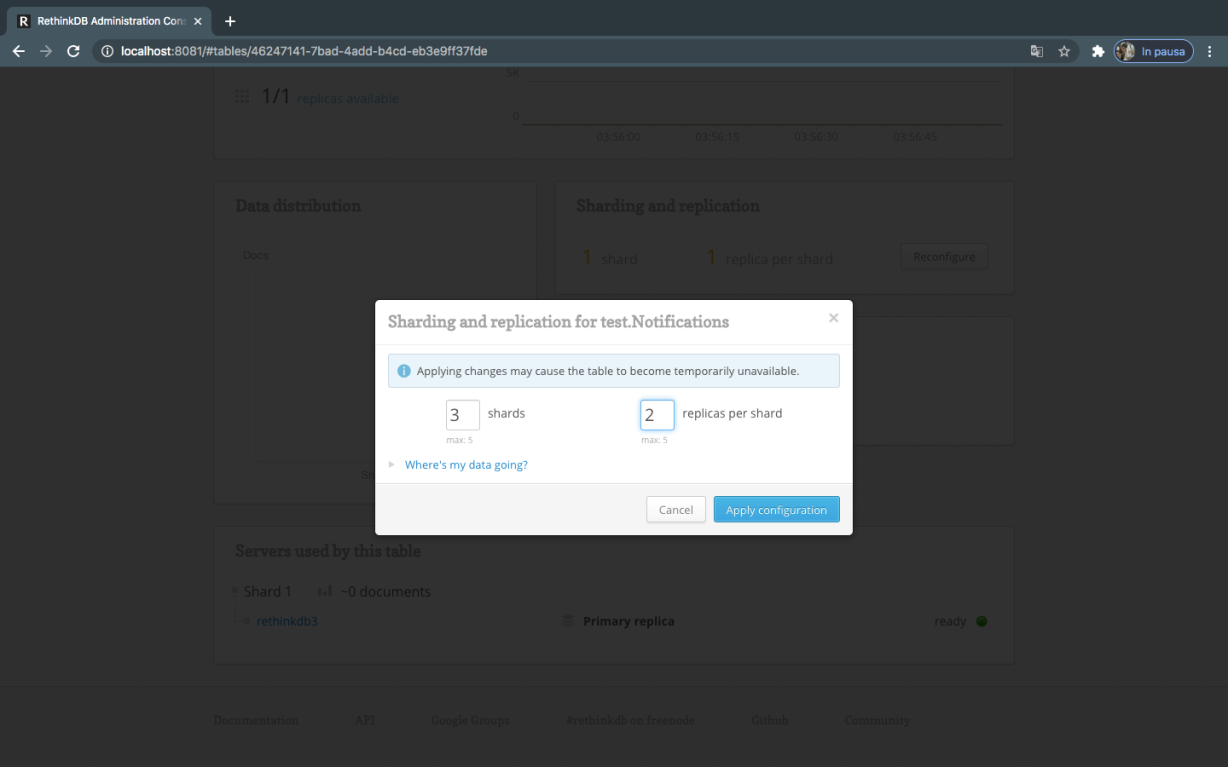


Figura 5.9

In seguito al “clic” sul pulsante “*Apply configuration”* i dati della tabella di sistema Notifications sono ridistribuiti come indicato nella figura 5.11

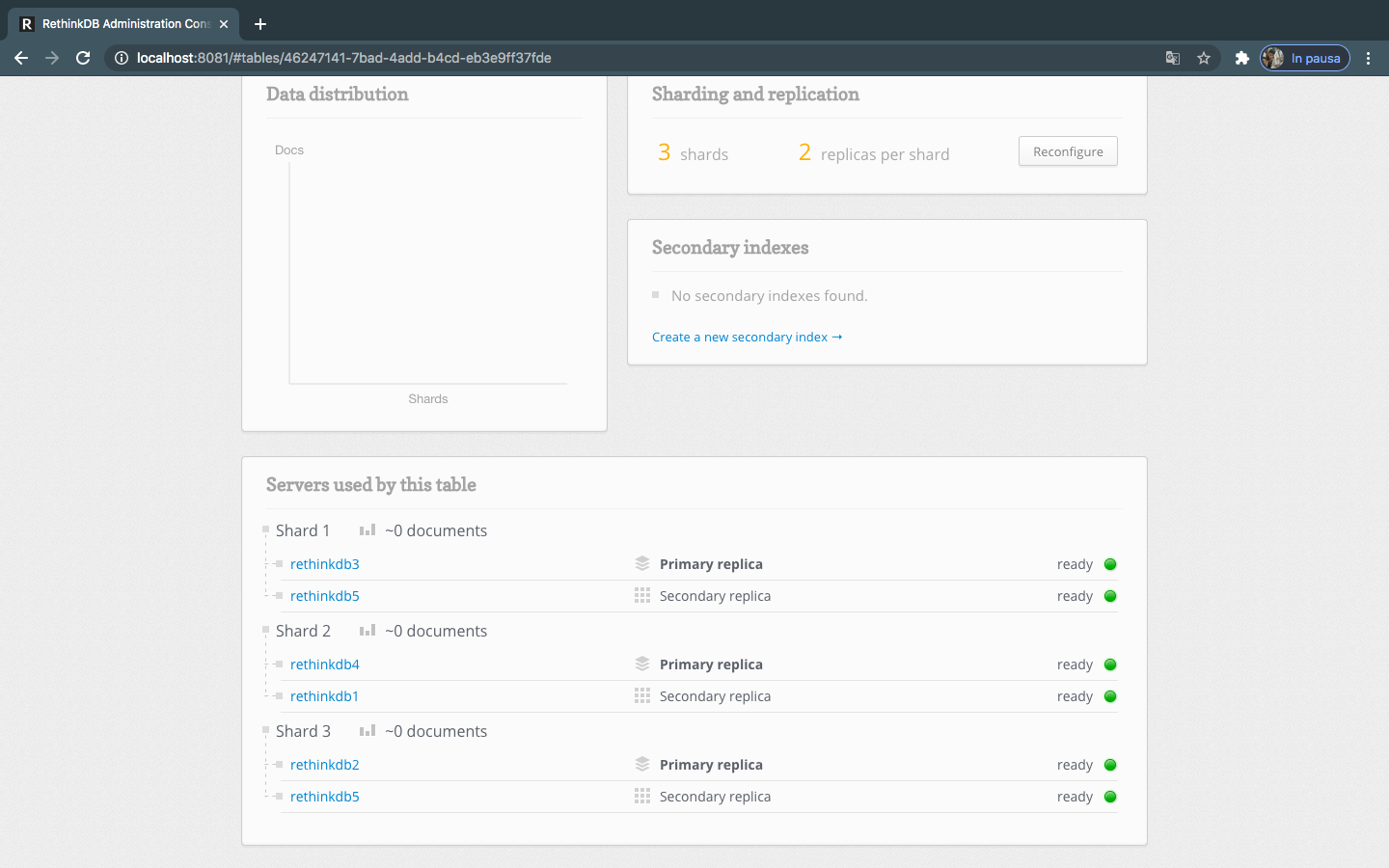


Figura 5.10

### Gestione con ReQL

Il linguaggio di interrogazione ReQL, tipico di RethinkDB, garantisce, oltre alle funzionalità di eseguire operazioni CRUD sui dati (*insert*, *delete* e *update*) e la funzionalità *changefeed* precedentemente discussa, un controllo dettagliato sul numero di repliche e la politica di Sharding adottata.

Se per l’implementazione del sistema fosse richiesto di gestire delle configurazioni più avanzate, come, ad esempio, decidere su quale specifico nodo del cluster inserire una delle repliche di un dato, allora, sarebbe stato necessario rivolgersi a comandi specifici offerti dal linguaggio ReQL. Quando si utilizza l’interfaccia grafica, infatti, i migliori punti di divisione dei dati sono autodeterminati da RethinkDB stesso.

Il sistema di tagging precedentemente discusso, può anche essere utilizzato, oltre che per la costruzione di un server RethinkDB a più nodi, anche per configurare le opzioni di Sharding e Replication. Nelle configurazioni di una tabella, in particolare, i tag permettono di archiviare su uno specifico nodo di interesse del cluster le repliche di un dato. Grazie a questo “sistema di tag”, quindi, è possibile avere un migliore controllo sulla locazione in cui saranno salvate le repliche dei dati di una tabella. Il comando ReQL a cui si fa riferimento è *reconfigure*.

Per assegnare 3 repliche della tabella users al nodo *us\_west* e 2 al nodo *us\_east*:



Figura 5.11

### Gestione nella libreria UtilityRethink

Anche attraverso la libreria di sistema UtilityRethink implementata è possibile gestire la Replication e lo Sharding dei dati di una tabella.

Come indicato in figura , attraverso una funzionalità offerta dal *DbManager*, vedi capitolo 3 sul design della libreria, l’utente è in grado di aggiornare lo sharding e il numero di repliche dei dati di una tabella appartenente a un server formato da più nodi.

Non avendo al momento la necessità di dover assegnare ad uno specifico nodo una determinata replica, il sistema lascia a RethinkDb stesso il compito di autodeterminarsi la migliore suddivisione dei dati.

ReconfigureTable

Figura 5.12

# **Conclusioni**

# **Bibliografia**

1. Database real-time. URL <https://medium.baqend.com/real-time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87>.
2. Database real-time. URL <https://en.wikipedia.org/wiki/Real-time_database>.
3. Design Publish/Subscribe. URL <https://it.wikipedia.org/wiki/Publish/subscribe>.
4. Firebase. URL <https://it.wikipedia.org/wiki/Firebase>.
5. Firebase Cloud Messaging. URL <https://en.wikipedia.org/wiki/Firebase_Cloud_Messaging#Web_Push_Support>.
6. Firebase Real-time Database. URL <https://firebase.google.com/docs/database>.
7. Firebase Real-time Database listener. URL <https://firebase.google.com/docs/database/admin/retrieve-data>.
8. Meteor. URL <https://en.wikipedia.org/wiki/Meteor_(web_framework)>.
9. Protocollo Meteor DDP. URL <https://github.com/meteor/meteor/blob/devel/packages/ddp/DDP.md>.
10. Parse. URL <https://en.wikipedia.org/wiki/Parse_(platform)>.
11. Parse. URL <https://medium.baqend.com/parse-is-gone-a-few-secrets-about-their-infrastructure-91b3ab2fcf71>.
12. Parse. URL <https://www.back4app.com/parse>.
13. Database auditing. URL <https://en.wikipedia.org/wiki/Database_audit>.
14. Meccanismi di auditing. URL [https://www.ugiss.org/2008/09/17/meccanismi-di- auditing/](https://www.ugiss.org/2008/09/17/meccanismi-di-   auditing/).
15. SQL Server Audit (Database Engine). URL <https://docs.microsoft.com/it-it/sql/relational-databases/security/auditing/sql-server-audit-database-engine?view=sql-server-ver15>.
16. Why you need a Database audit trail. URL <https://www.imperva.com/blog/why-you-need-a-database-audit-trail/>.
17. Esempio di software per il data-auditing. URL <https://www.datasunrise.com/blog/professional-info/why-you-need-a-database-audit-trail/>.
18. RabbitMQ. URL <https://it.wikipedia.org/wiki/RabbitMQ>.
19. Consuming changefeed. URL <https://github.com/bchavez/RethinkDb.Driver/wiki/Extra-C%23-Driver-Features#consuming-changefeeds>.
20. Esempio di utilizzo combinato delle libreria Bchavez/RethinkDb.driver e Reactive Extension. URL <https://github.com/bchavez/RethinkDb.Driver/blob/master/Source/RethinkDb.Driver.Tests/ReQL/ChangeFeedTests.cs>.
21. Strategia di Round Robin per la connessione al server RethinkDB. URL <https://github.com/bchavez/RethinkDb.Driver/wiki/Connections-&-Pooling#round-robin-strategy>.
22. RethinkDb, the open source database for the realtime web. URL <https://rethinkdb.com/>.
23. RethinkDb domande frequenti. URL <https://rethinkdb.com/faq>.
24. RethinkDB vs API di sincronizzazione in tempo reale (es Firebase) <https://rethinkdb.com/faq>
25. Linguaggio ReQL. URL <https://rethinkdb.com/docs/introduction-to-reql/>
26. Docker Desktop. URL <https://www.docker.com/products/docker-desktop>
27. I Docker container e le immagini. URL <https://www.docker.com/resources/what-container>
28. Docker Desktop for windows. URL <https://docs.docker.com/docker-for-windows/>
29. Bchavez/RethinkDb.driver. URL <https://github.com/bchavez/RethinkDb.Driver>
30. Reactive extensions. URL <http://reactivex.io/>
31. Simple injector. URL <https://simpleinjector.org/>
32. Metodologia di progettazione Scrum. URL <https://it.wikipedia.org/wiki/Scrum_(informatica)#:~:text=Scrum%20%C3%A8%20un%20framework%20agile,Ken%20Schwaber%20e%20Jeff%20Sutherland>.
33. Immagine RethinkDB utilizzata. URL <https://hub.docker.com/_/rethinkdb>
34. How to start a RethinkDB server. URL <https://rethinkdb.com/docs/start-a-server/>
35. Docker compose. URL <https://docs.docker.com/compose/>
36. Sharding e Replication su RethinkDB. URL <https://rethinkdb.com/docs/sharding-and-replication/>
37. Server tag di RethinkDB. URL <https://rethinkdb.com/docs/sharding-and-replication/#server-tags>

# **Ringraziamenti**