# 

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

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

**TITOLO DELL’ELABORATO**

*Elaborato in*

BASI DI DATI AVANZATE

*Relatore* *Presentata da*

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

Anno Accademico 2018 - 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:
* Capitolo 3:
* Capitolo 4:
* Capitolo 5:

**Indice**

[Sommario 3](#_Toc58752075)

[Specifiche e descrizione del problema 6](#_Toc58752076)

[Introduzione al problema 6](#_Toc58752077)

[Database-Realtime 6](#_Toc58752078)

[Publish/Subscribe 7](#_Toc58752079)

[Specifiche architetturali 8](#_Toc58752080)

[Specifiche funzionali 8](#_Toc58752081)

[Altri sistemi esistenti utilizzabili in futuro (Librerie e tecnologie) 9](#_Toc58752082)

[Firebase 9](#_Toc58752083)

[Database in tempo reale FireBase 10](#_Toc58752084)

[Meteor 10](#_Toc58752085)

[Parse 11](#_Toc58752086)

[RabbitMQ 13](#_Toc58752087)

[Conclusioni sulle altre tecnologie utilizzabili 14](#_Toc58752088)

[Bibliografia 15](#_Toc58752089)

**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* relazioni 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 ho 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 attraverso la libreria che ho implementato, contribuendo di conseguenza alla scalabilità del sistema.

Utilizzando questo pattern di messaggistica i servizi client del sistema riescono a registrarsi alla ricezione di notifiche utilizzando una funzionalità della libreria chiamata *UtilityRethink* che ho implementato (vedi capitolo *IRXNotifier*).

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 ho implementato coinvolge essenzialmente tre entità software principali: un applicativo client, una libreria e un server.

Il servizio client riesce a interagire con il server utilizzando la libreria.

I requisiti architetturali e tecnologici sono quindi:

• Architettura client - server;

• Accesso al server attraverso i protocolli http e https;

• Dbms non relazionale real-time

## Specifiche funzionali

La libreria di sistema implementata consente 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 che si riferiscono a eventi (tabella di sistema “Notifications”) con un certo valore di argomento (campo “arg”). Se ad esempio è inserita nella tabella “Notifications”, una nuova notifica con campo “arg” uguale a “A” allora saranno avvisati della variazione tutti i servizi interessati a notifiche aventi “arg” uguale a “A”.

A seguito della scelta di una tecnologia dbms per il server di sistema è possibile, inoltre, eseguire operazioni CRUD sulle tabelle che sono considerate di sistema dalla libreria implementata *UtilityRethinnk* (es. *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 gli altri sistemi dbms real-time disponibili in commercio: 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 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 suoi archivi di grandi dimensioni richiedono l’archiviazione 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*: database real-time o RabbitMQ. Nel mio progetto ho scelto di sviluppare il design del sistema indipendentemente dal tipo di tecnologia scelto per il server.

Se in futuro fosse, quindi, 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 successivo).

**Capitolo 3**

# Design libreria UtilityRethink

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

Come tecnologia server, per questo progetto, è stato scelto di utilizzare il sistema dbms non relazionale real-time chiamato RethinkDB che sarà analizzato in seguito.

La libreria di utility, implementata per garantire l’interfacciamento a RethinkDB, si chiama *UtilityRethink* è al momento disponibile solamente per applicativi scritti in linguaggio c# e sarà approfondita nei dettagli in questo capitolo.

## Aspetti statici

### Base Dati

Per la progettazione del database è stato necessario considerare l’utilizzo di un'unica tabella perché il sistema aveva come unico requisito di mantenere le notifiche relativi a eventi generati dagli applicativi aziendali. La tabella, in particolare, è stata chiamata *Notification* e dovrà raccogliere e mantenere ogni tipo di notifica di sistema. In tutte e tre i server RethinkDB implementati (uno, due, cinque nodi) e mostrati in seguito è stata inserita la tabella *Notification*.

Notifiche

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*.

L’unico vincolo impostato è che per inserire qualsiasi nuovo tipo di notifica in futuro su *Notifications* dovrà obbligatoriamente avere i campi della classe astratta *Notification* di *UtilityRethink* mostrata in figura 3.1.

(Foto Notification)

Figura 3.1

Ogni classe c# che rappresenta un tipo di notifica deve ereditare dalla classe base *Notification* sopra elencata in figura 3.1.

La libreria UtilityRethink offre due notifiche base implementate per il contesto aziendale e che ereditano di conseguenza dalla classe “Notification”: le *NotificationNewData* e le *NotificationExec*.

Se in futuro sarà necessario aggiungere qualche altro tipo di notifica sul database non sarà un problema. La classe c# che “mapperà” il nuovo tipo di notifica dovrà semplicemente ereditare dalla classe astratta *Notification*. Le funzionalità gestionali offerte dalla libreria UtilityRethink per le notifiche (vedi IQueryNotification e IRXNotifier), inoltre, lavorano con qualsiasi notifica ovvero ogni classe che eredita da “Notification”. Per implementare questa soluzione, ho utilizzato le proprietà di Reflection e dei generici.

Notifiche di nuovo dato

(foto Notifiche di nuovo dato)

Figura 3.2

Le notifiche di nuovo dato sono “mappate” nella libreria *UtilityRethink* come *NotificationNewData*.

Le notifiche di nuovo dato ereditando da *Notification* hanno quindi tutti i campi base delle notifiche e hanno un campo “Table” in più che servirà a identificare la tabella del database reale aziendale che è stata modificata. Queste notifiche quindi raccolgono su quale tabella è avvenuta una operazione: di insert, update o delete di un dato.

Grazie a questo tipo notifiche, che permettono di tracciare in tempo reale le operazioni svolte sulle tabelle del database reale, i servizi software aziendali saranno in grado di aggiornare i dati presenti sul sistema di cache aziendale.

Il sistema di caching è uno strato software che si 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 utilizzato per rispondere più rapidamente. Questo sistema viene, quindi, molto utilizzato dall’azienda per richiedere i dati in maniera più performante rispetto a interrogare direttamente il “database vero”.

Se per ottenere i dati, tutti i servizi client interrogassero il database reale sarebbe infatti molto dispendioso in termini di performance.

Notifiche di esecuzione

(Foto notifiche di esec)

Figura 3.3

Le notifiche di esecuzione sono invece chiamate dalla libreria *UtilityRethink* come *NotificationExec*.

Esse serviranno a notificare i servizi software aziendali interessati sullo stato di avanzamento di un Task/processo.

Come le notifiche di nuovo dato hanno tutti i campi base delle notifiche e hanno in aggiunta un campo idExec che rappresenta l’id del Task aziendale.

### Design UML applicativo di libreria UtilityRethink

#### Diagramma delle classi UtilityRethink

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

Figura 3.4

(cambiare mettendo le interfacce? A mio parere no perché sono le implementazioni a “comporsi” di altri oggetti)

#### UtilityRethink

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

Figura 3.5

E’ la classe fondamentale della libreria.

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

Figura 3.6

Ogni volta che la classe *UtilityRethink* è istanziata, viene creata una nuova connessione verso i nodi del server in ascolto sugli indirizzi precedentemente specificati (“indirizzoIp:numero Porta”).

In seguito alla connessione al server, *UtilityRethink* “connette” l’applicativo client che utilizza la libreria al database specificato. Nell’esempio indicato in figura 3.6, il database si chiama “test”. Se sul server rethink non esiste il database allora ne viene creato e su di esso vengono inserite tutte le tabelle che vengono considerate dalla libreria di “Sistema”. Per il contesto aziendale, l’unica tabella di sistema da considerare è la tabella delle che raccoglie le notifiche chiamata *Notifications*.

*UtilityRethink* consente all’utente di interagire con il database richiesto inizialmente attraverso i manager forniti dall’applicativo: *DbManager* e *INotificationsManager*.

In particolare, *INotificationsManager* è il manager della tabella di sistema *Notifications*.

Se in futuro sarà necessario avere altre tabelle di sistema basterà aggiungere a *UtilityRethink* il suo nuovo manager chiamato seguendo il formato “INomeTabellaManager”

Lo stesso oggetto che rappresenta la connessione al database richiesto generato in seguito all’istanziamento di UtilityRethink, viene inseguito utilizzato anche da tutti i sotto componenti della libreria per poter interagire con il server RethinkDB.

#### DbManager

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

Figura 3.7

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

Figura 3.8

Il *DbManager* è a tutti gli effetti il gestore del database specificato inizialmente durante l’istanziamento di *UtilityRethink*. Tra le funzionalità offerte troviamo la possibilità di creare e eliminare tabelle e indici.

Attenzione: Non è tuttavia possibile eliminare le tabelle di sistema del db (es. *Notification*).

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

#### NotificationsManager

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

Figura 3.9

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

Figura 3.10

Il *NotificationsManager* rappresenta, invece, il manager della tabella di sistema ”Notifications”. Come sottolineato precedentemente, se in futuro si avesse la necessità di avere altre nuove “tabelle di sistema”, si potrà creare semplicemente il suo nuovo manager per poi essere gestito dalla classe principale UtilityRethink.

*NotificationsManager* ha due sotto componenti fondamentali per la gestione delle notifiche: *IQueryNotification* e *IRXNotifier*.

#### IQueryNotification

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

# Figura 3.11

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

# Figura 3.12

Il *QueryNotification* è una funzionalità del *NotificationsManager* che permette all’utente di libreria di interrogare la tabella di sistema *Notifications*.

Offre le funzionalità di inserimento di nuove notifiche, cancellazione di notifiche aventi un determinato id e permette di richiedere notifiche con certi valori nei suoi campi.

(Utilizzo dei generici)

(Foto queryNotification per le Notifiche di nuovo dato)

(Foto queryNotification per le Notifiche di exec)

#### IRXNotifier

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

(Foto getter IRXNotifier)

IRXNotifier è il notificatore di eventi.

Offre le proprie funzionalità sempre per la tabella di sistema “Notifications”.

Grazie agli approfondimenti svolti durante lo sviluppo dell’applicativo sul pattern observer e ad una attento utilizzo delle librerie “bchavez/RethinkDb.Driver” e Reactive Extensions (Rx) è stato possibile implementare il notificatore [[22]](#Consuming_changefeed) .

Il notificatore rimane in ascolto ad eventi sulla tabella “Notifications”.

In particolare, può intercettare eventi che riguardano solamente notifiche che abbiano certi valori di argomento (campo “arg”).

Gli eventi possono essere inserimento, cancellazione, modifica di una notifica che abbia come argomento uno tra quelli presi in considerazione.

In seguito alla “richiesta di registrazione” viene restituito un NotificationSubscription, oggetto che raccoglie l’observable con un suo id.

Ora, passaggio fondamentale, chi utilizza la libreria dovrà sottoscrivere l’observable a dei metodi implementati in base alle proprie esigenze.

Per esempio come mostrato in figura, in caso di intercettazione corretta di un evento verrà chiamato il metodo onNext su cui è possibile manipolare la nuova Notifica oppure la vecchia (in caso di update).

In caso di errore verrà chiamato il metodo OnError mentre al termine dell’ascolto (se verrà chiamato il metodo StopListening sull’observable) verrà invocato il metodo OnComplete.

I tre metodi, OnNext, OnError e OnComplete verrano in questo modo implementati lato cliente ognuno in base alle proprie esigenze [[23]](#notifier_es) .

Esempio di utilizzo notificatore sulle notifiche di esecuzione:

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

(x ha in se lo stato dell’evento catturato: nuovo Dato, vecchio Dato, ecc …)

Per ottenere un notificatore di qualsiasi altro tipo di notifica, sempre a patto che erediti dalla classe base “Notification.cs”, come ad esempio le notifiche di nuovo dato (“NotificationNewData”), è sufficiente sostuire dentro le “<>” “NotificationNewData” al posto di “NotificationExec”.

##### Esempio di OnNext, OnError e OnComplete

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

#### Connection

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

L’ oggetto ConnectionNodes viene istanziato inizialmente da UtilityRethink per poi essere utilizzato da tutti i sotto-componenti (IQueryNotifications e IRXNotifier).

I sotto-componenti, infatti, per interfacciarsi al server RethinkDb utilizzano quindi la stessa istanza di ConnectionNodes.

ConnectionNodes tiene traccia dei nodi del server a cui è connesso tramite i DbOptions.

In questo caso, in particolare, possono essere 1, 2 o 5, in base al numero dei nodi del server su cui ci si è connessi.

Il campo “conn” di ConnectionNodes rappresenta il vero oggetto di connessione al database.

Esso viene istanziato solamente alla prima chiamata del metodo GetConnection(), ovvero alla prima interazione col server RethinkDB.

Alle successive chiamate di GetConnection, da parte dei sotto componenti dei vari Manager non viene più istanziato ma viene sempre restituito lo stesso oggetto.

Riinizializzare più volte “conn” risulterebbe infatti un’operazione pesante.

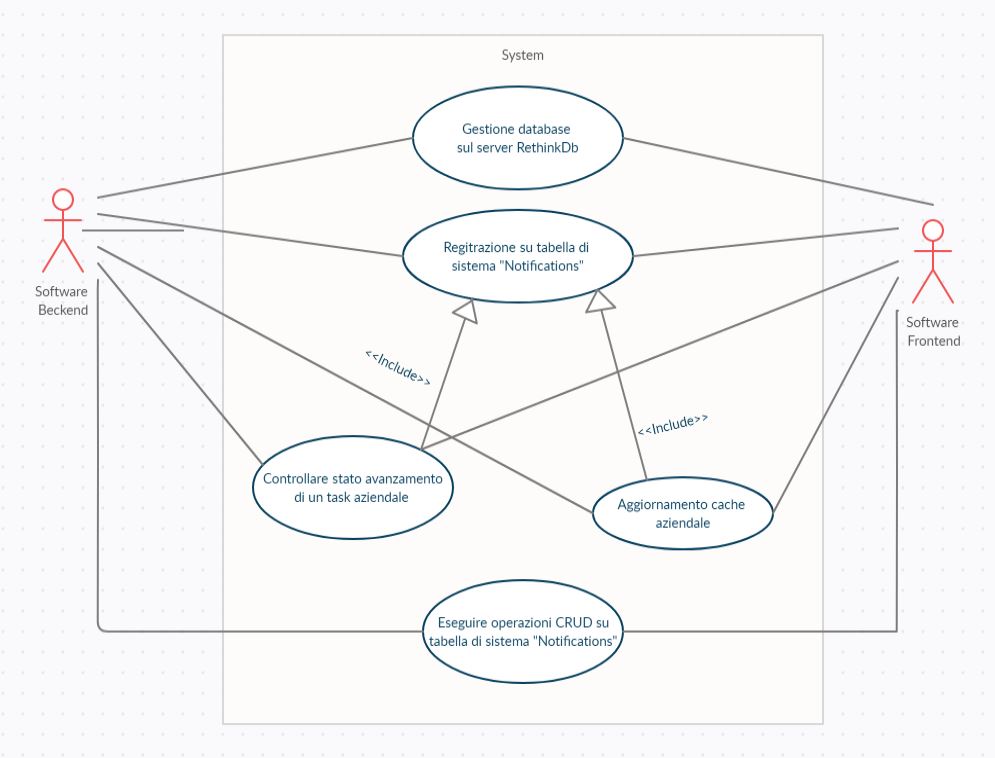
La connessione al server Rethink viene eseguita grazie ad un’algoritmo di RoundRobin, implementato dalla libreria “Bchavez/Rethinkdb”, che sceglie a run-time un nodo disponibile al momento.

ConnectionNodes gestisce, infine, anche i tentativi di connessione su indirizzi Ip non raggiungibili [[24]](#connessione_roundRobin) .

Se dopo 20 secondi non si riesce a connettere al server viene fatto scattare il Timeout per la mancata connessione.

## Aspetti dinamici

### Caso d’uso aziendale



#### Simple Injector

Nel codice dei servizi software di frontend e beckend aziendali serviva la possibilità di richiamare la stessa istanza della libreria UtilityRethink in diversi punti del codice.

Era inoltre necessario fornire un punto di accesso globale a tale istanza.

A tal proposito, ho approfondito i design patterns Singleton e Factory method per poter utilizzare correttamente la libreria Simple Injector in .Net.

Attraverso la classe container, di Simple Injector, è infatti possibile registrare un’istanza di una classe come se fosse un “singleton”.

Successivamente alla registrazione l’istanza registrata può essere richiamata in diversi punti del codice.

Ho sfruttato, quindi, questa caratteristica della libreria Simple Injector per simulare il pattern Singleton sulla classe principale della libreria “UtilityRethink.cs”.

# 

### Diagrammi di attività

(foto diagrammi attivita)

**Capitolo 6**

# 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>