# 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

**Indice**

[Introduzione 3](#_Toc57662273)

[Azienda 3](#_Toc57662274)

[Problema Affrontato 3](#_Toc57662275)

[Auditing e notifiche di eventi 4](#_Toc57662276)

[Specifiche funzionali e architetturali 4](#_Toc57662277)

[Altri sistemi esistenti 5](#_Toc57662278)

[Librerie e tecnologie 5](#_Toc57662279)

[I Trigger 5](#_Toc57662280)

[Tecniche di auding 5](#_Toc57662281)

[Scelte architetturali 6](#_Toc57662282)

[Software 6](#_Toc57662283)

[Librerie 7](#_Toc57662284)

[Metodologia di Progettazione 8](#_Toc57662285)

[Implementazione Server 9](#_Toc57662286)

[Sharding e Replication 10](#_Toc57662287)

[Gestione con Interfaccia Web 11](#_Toc57662288)

[Gestione con ReQL 12](#_Toc57662289)

[Gestione nella libreria 13](#_Toc57662290)

[Comandi per la gestione del server 13](#_Toc57662291)

[Test e Performance 14](#_Toc57662292)

[Risultati Test Performance 15](#_Toc57662293)

[Server Rethink a singolo nodo 15](#_Toc57662294)

[Server MySQL a singolo nodo 15](#_Toc57662295)

[Server Rethink a cinque nodi 16](#_Toc57662296)

[Una shard con cinque repliche ciascuna 16](#_Toc57662297)

[Cinque shard con una sola repliche 16](#_Toc57662298)

[Due shard con tre repliche ciascuna 16](#_Toc57662299)

[Design libreria UtilityRethink 17](#_Toc57662300)

[Aspetti statici 17](#_Toc57662301)

[Base Dati 17](#_Toc57662302)

[Notifiche 17](#_Toc57662303)

[Notifiche di nuovo dato 19](#_Toc57662304)

[Notifiche di esecuzione 19](#_Toc57662305)

[Casi d’uso 20](#_Toc57662306)

[Simple Injector 20](#_Toc57662307)

[Design UML applicativo 21](#_Toc57662308)

[Design della Struttura generale UML libreria UtilityRethink 21](#_Toc57662309)

[UtilityRethink 22](#_Toc57662310)

[DbManager 23](#_Toc57662311)

[NotificationsManager 24](#_Toc57662312)

[IQueryNotification 24](#_Toc57662313)

[IRXNotifier 25](#_Toc57662314)

[Esempio di OnNext, OnError e OnComplete 27](#_Toc57662315)

[Connection 28](#_Toc57662316)

[Aspetti dinamici 29](#_Toc57662317)

[Diagrammi di attività 29](#_Toc57662318)

[Sviluppi futuri 30](#_Toc57662319)

[Conclusioni 31](#_Toc57662320)

[Bibliografia 32](#_Toc57662321)

**Capitolo 1**

# Introduzione

## Azienda

Il tirocinio è stato svolto presso l’azienda Energy Software a Faenza. EnergySoftware s.r.l è un’azienda che nasce nel 2015 da un team di persone con esperienza decennale con l’obiettivo di fornire tutta una serie di servizi legati al mondo dell’informatica tra cui sviluppo di software per i mercati energetici, sviluppo

di applicativi per specifiche esigenze del cliente, consulenza sistematica

specializzata su server e networking.

## Problema Affrontato

Il lavoro prevede il design e la realizzazione di un sistema di notifiche che permetta di distribuire eventi generati su software di beckend e di frontend in modo da fornire informazioni costantemente aggiornate agli utenti delle applicazioni web.

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

La libreria è stata prototipata in C# su piattaforma .NET Core e il codice prodotto è stato organizzato in maniera efficiente e riutilizzabile nei prodotti della società.

Dal punto di vista del database, abbiamo studiato lo schema migliore di rappresentazione dei dati per ottenere le migliori prestazioni possibili a fronte delle necessarie caratteristiche di consistenza ed isolamento.

I risultati ottenuti sono stati oggetto di misurazione e presentazione.

## Auditing e notifiche di eventi

blablablablablablablablablablablablablablablablabla

## Specifiche funzionali e architetturali

Descrizione delle funzionalità (funzionali) + tipo di architettura richiesta (client server, interfaccia web….)

Vedi ad esempio in questo capitolato cosa si intende per requisiti architetturali (<https://www.lilt.it/sites/default/files/allegati/amm-trasparente/2018-11/capitolato-speciale-all.1-specifiche-tecniche.pdf>)

**Capitolo 2**

# Altri sistemi esistenti

## Librerie e tecnologie

blablablablablablablablablablablablablablablablablabla

## I Trigger

I sistemi relazionali blablablablablablablablablablablablablablablablablabla

## Tecniche di auding

Vedi database auding wikipedia :

(<https://www.lilt.it/sites/default/files/allegati/amm-trasparente/2018-11/capitolato-speciale-all.1-specifiche-tecniche.pdf>)

(non andare troppo nello specifico sulle altre tecnologie)

**Capitolo 3**

# Scelte architetturali

## Software

Inizialmente, ho ricercato il miglior software utilizzabile per l’applicativo.

Il progetto è stato implementato sull’ambiente di sviluppo Visual Studio.

Abbiamo pensato, per le esigenze del progetto, di utilizzare RethinkDb, un dbms non relazionale di tipo documentale.

(cosa significa db documentale -> le tabelle non seguono uno schema, i documenti di una stessa tabella possono avere campi diversi)

Per interrogare un server RethinkDb viene utilizzato il linguaggio ReQl.

Il Reql è un linguaggio di interrogazione molto simile a SQL classico.

Permette, infatti, di eseguire operazioni di “join” tra tabelle di uno stesso database.

E’ inoltre possibile generare una Pipeline di operazioni concatenandole una dopo l’altra separate da un punto.

(Mettere es Query Reql: )

Tra i software largamente utilizzati troviamo, inoltre, Git e Docker Desktop.

Git è stato fondamentale alla gestione del repository di progetto.

Docker Desktop, invece, è stato utile alla creazione e mantenimento del server Rethinkdb.

(Docker container, cosa sono ?)

## Librerie

Tramite lo store NuGet di Visual Studio è possibile scaricare diverse librerie di supporto ai propri progetti.

Ho utilizzato le Librerie bchavez/ RethinkDb.Driver, Reactive Extensions (Rx) e Simple Injector.

“bchavez/ RethinkDb.Driver” è un driver che fornisce un’api in .Net, utile per l’interfacciamento al server Rethinkdb.

Pur non essendo un driver “ufficiale”, è molto simile a quello ufficiale scritto in java.

Per implementare il manager delle notifiche è stata utilizzata la libreria Reactive Extensions (Rx).

Reactive Extensions (Rx) è una libreria per la composizione di programmi asincroni e basati su eventi utilizzando sequenze osservabili e operatori di query in stile LINQ.

Simple Injector, infine, sarà utilizzata dai sofware di frontend e beckend aziendali per gestire e richiamare le funzionalità dell’ applicativo.

Grazie ad essa, è infatti possibile registrare un’istanza di una classe e fare in modo che quella restituita sia sempre la stessa al momento della registrazione.

Attraverso questo meccanismo di Simple Injector è stato possibile implementare automaticamente il Pattern Singleton sulla classe principale della libreria (UtilityRethink).

## Metodologia di Progettazione

In seguito alla scelta del software, è stato scelto Scrum come metodologia di progettazione agile.

Esso permette la divisione del progetto in Sprint, ovvero intervalli di tempo di durata fissa generalmente da uno a quattro settimane.

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 più cose nuove che entrano nella lista “ToDo”.

A tal proposito, per la gestione del progetto ho utilizzato Jira Software attraverso credenziali datemi dall’azienda.

Quando iniziavo una nuova attività (Sprint) essa passava dallo stato “ToDo” al “Doing” per poi andare in “Done” quando veniva completato.

## Implementazione Server

Per poter testare le capacità del dbms RethinkDb abbiamo pensato di affidarci alla tecnologia offerta dai container docker.

Attraverso il sito Docker-hub, sito web su cui è possibile cercare e scaricare immagini docker di applicativi, abbiamo scelto di utilizzare l’immagine ufficiale “rethinkdb” mantenuta dalla società RethinkDb.

In seguito al download dell’immagine, utilizzando sul terminale i comandi di Docker-compose sono riuscito a costruire tre diversi tipi di server Rethink.

Server a singolo nodo

(foto terminale e interfaccia rethink)

Server a due nodi

(foto terminale e interfaccia rethink)

Server a cinque nodi

(foto terminale e interfaccia rethink)

Ogni nodo del server rappresenta di fatto un container.

Nei due casi del server a più nodi, attraverso lo script bash “start-rethinkdb-cluster.sh” è stato possibile eseguire un join dei nodi-container in unico server.

Attraverso questa tecnica, i nodi del server collaborano tra loro per rispondere alle richieste in arrivo e anche per poter sfruttare le politiche di Replication e di Sharding.

Bisogna specificare che, in questo caso, il server risiede completamente sulla macchina locale avendo utilizzato i comandi di Docker-compose.

In un contesto più articolato, se necessario, è possibile utilizzare altre tecnologie per distribuire i diversi nodi del server su più macchine fisiche.

Alcune tecnologie di riferimento per questo contesto sarebbero ad esempio Docker Swarm o Kubernetes.

### Sharding e Replication

Due aspetti che bisogna considerare, durante lo sviluppo di un cluster, sono le politiche di Sharding e Replication sui dati.

In particolare, le politiche di Sharding servono a distribuire i dati al meglio sul Cluster.

I dati vengono inizialmente suddivisi in piccole parti, dette shards, per essere successivamente inserite al meglio nei diversi nodi del server.

Una buona strategia di sharding è fondamentale per ottimizzare le performance.

Le diverse partizioni dei dati, inseguito ad una richiesta, potranno essere lette in parallelo dai diversi nodi.

(Strategia di sharding mio caso:

vedi paragrafo **Sharding e replica tramite ReQL:**

<https://rethinkdb.com/docs/sharding-and-replication/> li dice che usa politica di hash per lo sharding -> non ho controllo sulla partizione)

La Replication, invece, serve alla creazione e gestione di copie dei dati sui nodi del Cluster.

In caso di “node failure” è una tecnica che previene la perdita dei dati nel caso siano stati precedentemente salvati anche su un secondo nodo.

Attraverso la Replication, inoltre, un dato può essere letto in parallelo da diversi utenti siccome è presente in più nodi del Cluster.

Tramite la replicazione, è inoltre 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, infatti, altà probabilità che risieda su un nodo particolarmente lontano rispetto a dove è stato richiesto.

Ci sarebbero, di conseguenza, alti costi di rete.

(Strategia di Replication da me usata)

#### Gestione con Interfaccia Web

RethinkDB consente di suddividere e replicare il cluster in base ad una tabella.

Le impostazioni possono essere controllate facilmente dalla console di amministrazione web.

Quando si utilizza l'interfaccia utente Web, è sufficiente specificare il numero di frammenti desiderati e, in base ai dati disponibili, è RethinkDB che determinerà i migliori punti di divisione per mantenere i frammenti bilanciati.

Per dividere i propri dati:

Vai alla vista tabella ( *Tabelle* → *nome tabella* ).

Fare clic sul pulsante *Riconfigura* .

Impostare il numero di frammenti e repliche che si desidera.

Fare clic sul pulsante *Applica configurazione*

(Foto Reconfigure tabella Notifications table overview)

#### Gestione con ReQL

Il linguaggio ReQL consente anche un controllo più dettagliato sulla replica di un dato.

E’ possibile, quindi, oltre all’interfaccia grafica, gestire le repliche utilizzando comandi forniti dal linguaggio ReQL.

Se si avesse la necessità di ottenere configurazioni più avanzate, (es decidere su quale nodo di un Cluster inserire una replica di una tabella), sarebbe necessario rivolgersi a comandi del linguaggio ReQL.

Se si utilizza l’interfaccia grafica vengono infatti autodeterminati i migliori punti di divisione.

Per raggruppare i nodi nei cluster, RethinkDb utilizza i tag di ciascun nodo definiti dall’utente.

A tutti i nodi in un cluster RethinkDB possono essere assegnati zero o più tag:

rethinkdb --server-tag us\_west

I tag possono anche essere utilizzati nelle configurazioni di tabella per mappare le repliche ai server specificati dal tag utilizzando il comando ReQL reconfigure.

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

r**.**table('users')**.**reconfigure(shards**=**2, replicas**=**{'us\_west':3,

'us\_east':2}, primary\_replica\_tag**=**'us\_east')**.**run(conn)

#### Gestione nella libreria

Nella libreria UtilityRethink implementata, un suo sottocomponente (il DbManager) ha anch’esso una funzionalità di reconfigure che permetterà di assegnare il numero di shard e repliche dei dati di una tabella.

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

### Comandi per la gestione del server

Aprire il terminale e dirigersi sulla cartella di progetto corrispondente all’opzione del server scelta (uno, due o cinque nodi).

In seguito, scrivere sul terminale: “docker-compose -f docker-compose.yml build”.

Questo comando è necessario eseguirlo solamente al primo utilizzo del tipo di server scelto.

In seguito all’esecuzione, infatti, viene costruita e salvata permanentemente sul proprio dispositivo l’immagine docker del server RethinkDb (se necessario è possibile successivamente cancellare l’immagine, vedi documentazione Docker).

Successivamente eseguendo, sempre su terminale, il comando “docker-compose -f docker-compose.yml up -d” il server Rethink è online e i suoi nodi sono in attesa di ricevere richieste.

Per stoppare il server digitare “docker-compose -f docker-compose.yml stop”.

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

Se, ad esempio, inserissi dei nuovi dati sul db e stoppassi successivamente il server, al prossimo comando di up il server mantiene i dati precedentemente caricati.

I tre comandi elencati valgono per tutti e tre i tipi di server (1, 2, 5 nodi).

## Test e Performance

Il test sulle performance riguarda uno studio del comportamento del dbms RethinkDb in termini di velocità di lettura e scrittura.

Siccome attraverso l'interfaccia utente Web, è possibile gestire semplicemente le politiche di Sharding e Replication, nei due casi in cui il server è composto da più nodi ho testato le performance e il comportamento del dbms tenendo anche in considerazione questi due aspetti.

Ricordo che il server, in questo test, risiede interamente sulla macchina locale avendo utilizzato i comandi di Docker-compose.

Per uno studio completo del comportamento e delle performance bisognerebbe oltre ai casi analizzati distribuire anche i vari nodi del server Rethink (nei casi a più nodi) su diverse macchine.

Ho progettato un piccolo applicativo scritto in .Net finalizzato alla fase di test.

In particolare sui tre server Rethink ho creato due tabelle: “Author” e “Post” all’interno di un database “test”.

(Foto db con tabelle Author e Post)

Ho inserito circa 200 documenti su Author e 20 mila su Post.

Gli autori sono caratterizzati da: id, nome, età, hobby. Mentre i post da: id, author\_id, titolo, contenuto.

Da ogni post è possibile ricavare il suo autore attraverso il campo “author\_id” .

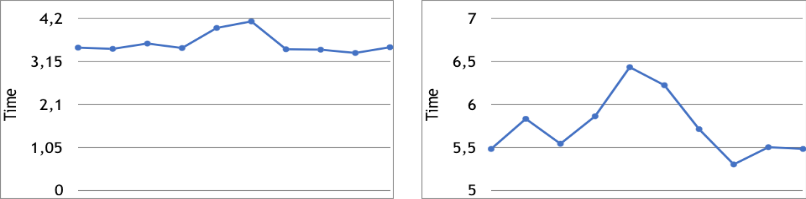
Ho costruito quindi un indice secondario sul campo “author\_id” di “Post” per ottenere le migliori performance possibili.

Nei casi a 1 e 5 nodi del server ho misurato in secondi quanto tempo impiega Rethink ad effettuare 50 inserimenti su Post e una query di selezione basata tra un join tra Author e Post per riuscire a determinare quanti post ha scritto ogni autore sfruttando quindi l’indice sul campo “author\_id” su “Post” precedentemente costruito.

### Risultati Test Performance

Server Rethink a singolo nodo

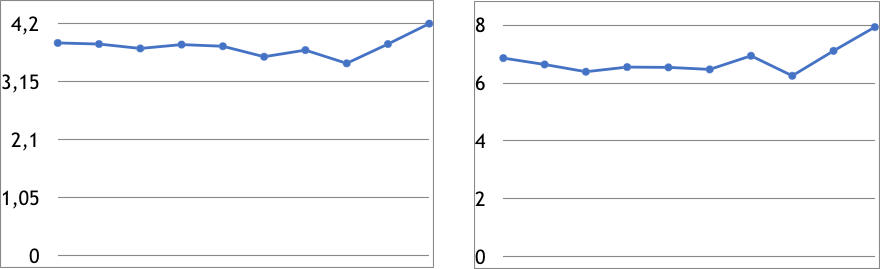
Singolo nodo (non è possibile applicare politiche di sharding e replication):

Select (media: 3,577 s) Insert (media: 5,735 s)

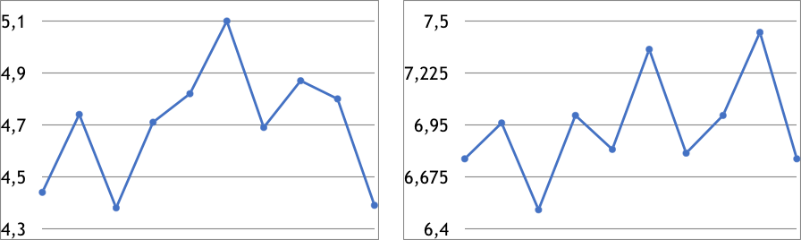
#### Server MySQL a singolo nodo

Server Rethink a cinque nodi

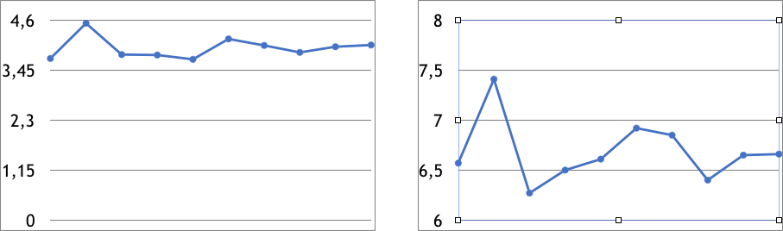
Una shard con cinque repliche ciascuna

Select (media: 3.787 s) Insert (media: 6,768 s)

Cinque shard con una sola repliche

Select (media: 4,694 s) Insert (media: 6,941 s)

Due shard con tre repliche ciascuna

Select (media: 3,963 s) Insert (media: 6,684 s)

**Capitolo 4**

# Design libreria UtilityRethink

Successivamente allo studio del dbms RethinkDb, il lavoro ha previsto l’implementazione di una libreria di utility chiamata UtilityRethink.

UtilityRethink è stata scritta in .Net per permettere ai software di beckend e frontend aziendali di interrogare i server RethinkDb.

## Aspetti statici

### Base Dati

Siccome nel contesto aziendale fosse necessario mantenere le notifiche di eventi generati dai propri software di frontend e beckend è stato necessario considerare solamente l’utilizzo di una tabella.

Ho inserito quindi nei tre server implementati una tabella Notification che raccogliesse

tutte le possibili notifiche aziendali.

Notifiche

Rethinkdb essendo un db di tipo non relazionale accetta la diversità di campi tra documenti di una stessa tabella.

Diversamente dai dbms relazionali, RethinkDb non segue uno schema fisso per le proprie tabelle

Ho sfruttato quindi questa caratteristica per raccogliere due diversi tipi di notifiche all’interno della stessa tabella “Notifications”.

Qualsiasi tipo di notifica presente sulla “Notification”, per mia imposizione dovrà avere i campi della classe astratta “Notifications”

(Foto Notifications)

Dal punto di vista di UtilityRethink (lato applicativo), ogni classe che rappresenta un tipo di notifica deve ereditare dalla classe base Notification sopra elencata.

Grazie a questo design della libreria, se in futuro sarà necessario aggiungere qualche altro tipo di notifica non sarà un problema.

La classe c# che “mapperà” il nuovo tipo di notifica dovrà ereditare dalla classe astratta Notifications.

Le funzionalità gestionali delle notifiche offerte da UtilityRethink, (vedi IQueryNotification e IRXNotifier), inoltre, lavorano con qualsiasi tipo di notifica sempre a patto che, la classe della notifica, erediti da “Notification”.

Per implementare questa soluzione, ho utilizzato la proprietà della Reflection e dei generici. La reflection, in particolare è servita per riuscire a determinare a run-time il tipo di una notifica (campo “type” di “Notification).

Nel contesto aziendale esistono, in particolare, due diversi tipi di notifiche.

Le notifiche di nuovo nuovo dato, e le notifiche di esecuzione.

Notifiche di nuovo dato

(foto Notifiche di nuovo dato)

Le notifiche di nuovo dato del db sono “mappate” dalla libreria UtilityRethink come “NotificationNewData”.

Le notifiche di nuovo dato hanno tutti i campi “base” delle notifiche e hanno, in aggiunta, un campo “Table”.

“Table” servirà a identificare la tabella del db reale aziendale modificata, ad esempio in seguito ad un’operazione di insert o update di un dato.

In questo modo, in seguito, sarà possibile aggiornare anche la “cache del sistema aziendale” su cui vengono effettuate le ricerche.

Il sistema di caching aziendale viene molto utilizzato dall’azienda per richiedere i dati in maniera più performante rispetto a interrogare direttamente il “db vero”.

Se per ottenere i dati, tutti interrogassero il db reale sarebbe infatti molto dispendioso.

Notifiche di esecuzione

(Foto notifiche di esec)

Le notifiche di esecuzione, invece, sono “mappate” da UtilityRethink come “NotificationExec”.

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

Hanno tutti i campi base delle notifiche e in aggiunta un campo idExec che rappresenta l’id del Task aziendale

### Casi d’uso

(Foto diagramma casi d’uso)

#### 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.

Ho studiato e approfondito a tal scopo i pattern Singleton e Factory method per un’utilizzo corretto della libreria Simple Injector in .Net.

Attraverso la classe container, questa libreria, infatti, permette la registrazione di un’istanza di una classe.

Successivamente alla registrazione l’istanza registrata può essere richiesta 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”.

# 

### Design UML applicativo

#### Design della Struttura generale UML libreria UtilityRethink

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

(cambiare mettendo le interfacce?)

#### UtilityRethink

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

E’ la classe fondamentale della libreria.

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

Una volta istanziato crea una nuova connessione con un db sul Cluster Rethink (se non esiste viene creato) e su di esso crea (sempre se non esiste gia quel db) tutte le tabelle di “Sistema”.

UtilityRethink consente all’utente di interagire con il db richiesto sul cluster attraverso i due manager dell’applicativo: DbManager e INotificationsManager

In questo caso, nel nostro caso d’uso, l’unica tabella di sistema è “Notifications” perché abbiamo solamente il manager delle Notifiche “INotificationsManager”.

L’oggetto “connessione” creato viene inseguito “passato” a tutti i sotto-componenti, ovvero i manager, per poter essere utilizzato.

#### DbManager

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

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

E’ a tutti gli effetti il manager del Db, possibile creare e eliminare tabelle indici.

Attenzione: Non è possibile eliminare le tabelle di sistema (es “Notification”).

#### NotificationsManager

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

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

E’ il manager delle notifiche.

Essendo il manager delle notifiche la sua tabella di sistema è “Notification”.

Se in futuro si avrà la necessità di avere altri Manager per altre tabelle si potrà gestire in maniera semplice grazie a questo design andando ad aggiungerli a Utility Rethink.

Ha due sotto componenti sempre relativi alle notifiche: 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

Permette di effettuare le “interrogazioni” alla tabella di sistema “Notifications”.

operazioni CRUD.

Inserimento di nuove notifiche, cancellare notifiche con un certo id e richiedere Notifiche in base a determinati valori dei suoi campi.

#### IRXNotifier

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

RXNotifier è un notificatore di eventi, sulla tabella di sistema di INotification, quindi su “Notifications”.

Grazie agli studi svolti durante l’attività “Notifiche”, allo studio del pattern observer e alla combinazione delle librerie “bchavez/RethinkDb.Driver” e Reactive Extensions (Rx) è stato possibile implementare il notificatore.

Il notificatore rimane in ascolto sulla tabella “Notifications” e, in particolare, può anche 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.

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 le notifiche di nuovo dato (“NotificationNewData”) basta 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

Oggetto Connection viene istanziato all’inizio dall’UtilityRethink per poi passarlo a tutti i sotto-componenti dell’applicativo. Tutti i sotto-componenti per interfacciarsi al Cluster Rethink utilizzano quindi lo stesso oggetto Connection.

Nello specifico il campo conn che rappresenta la connessione vera e propria viene istanziata solamente alla prima interazione col db, successivamente non viene più istanziato ma viene sempre restituito lo stesso oggetto perché altrimenti sarebbe un’operazione pesante.

ConnectionNodes accetta una lista di DbOptions.

Nel nostro caso, in particolare, possono essere 1, 2 o 5, in base al numero dei nodi del Cluster su cui connettersi.

La connessione viene eseguita tramite un’algoritmo di RoundRobin che sceglie a run-time il nodo disponibile al momento.

Se ci si connette su indirizzi ip non raggiungibili dopo 20 secondi viene fatto scattare il Timeout per la mancata connessione.

## Aspetti dinamici

### Diagrammi di attività

(foto diagrammi attivita)

**Capitolo 5**

# Sviluppi futuri

**Capitolo 6**

# Conclusioni

**Capitolo 6**

# Bibliografia