

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

Francesco Foschini

15 gennaio 2022

## Sommario

L'obiettivo di questo studio prevede la realizzazione di un sistema che permetta di emettere e distribuire notifiche legate ad eventi. Questa funzionalità sarà fruita da servizi software al fine di fornire informazioni costantemente aggiornate agli utenti delle applicazioni web e per monitorare lo stato di avanzamento di task/procedure. La parte iniziale del lavoro è stata dedicata all'analisi del contesto in cui si colloca questo problema: è stato definito il concetto di Database real-time e di seguito sono stati analizzati i sistemi e le tecnologie disponibili in commercio in grado di offrire il servizio di distribuzione delle notifiche ai servizi client implementando il famoso pattern di messaggistica *Publish/Subscribe*. Per il design del sistema è stato, quindi, 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 due attività principali: lo sviluppo del server di sistema e la progettazione e implementazione di una libreria. In seguito all'analisi del software scelto per il server di sistema (RethinkDB) si è proceduto con un'attenta fase di progettazione della libreria di sistema *UtilityRethink* che garantirà ai servizi client l'interfacciamento con il server RethinkDB. Grazie a questa iniziale fase di analisi e progettazione si è potuto sviluppare un primo prototipo del server e implementare una libreria in c funzionante ed efficiente che garantisca il principale requisito del sistema. I servizi client attraverso la libreria e interagendo con il server RethinkDB potranno, infatti, sia inviare nuove notifiche, sia registrarsi per ricevere quelle per cui sono interessati. IL TESTING DELLA LIBRERIA E' STATO EFFETTUATO ATTRAVERSO L'IMPLEMENTAZIONE DI UN PICCOLO PROGETTO DI TESTING E L'IMPLEMENTAZIONE DI UNA DEMO.

# Indice

<b>1 Specifiche e descrizione del problema</b>	<b>4</b>
1.1 Database-Realtime . . . . .	4
1.2 Publish/Subscribe . . . . .	5
1.3 Specifiche architetturali . . . . .	5
1.4 Specifiche funzionali . . . . .	7
<b>2 Tecnologia Scelta</b>	<b>8</b>
2.1 RethinkDb . . . . .	8
2.2 ReQL . . . . .	10
<b>3 Progettazione libreria UtilityRethink</b>	<b>12</b>
3.1 Diagramma dei casi d'uso . . . . .	12
3.2 Diagramma delle attività e di sequenza . . . . .	13
3.2.1 Come sono distribuite le notifiche ai client . . . . .	14
3.3 Base Dati . . . . .	15
3.4 Progettazione notifiche . . . . .	16
<b>4 Implementazione e testing della libreria UtilityRethink</b>	<b>17</b>
4.1 Software utilizzati . . . . .	17
4.2 Diagrammi delle classi dei componenti di libreria . . . . .	17
4.2.1 Notifiche . . . . .	18
4.2.2 Notifiche di nuovo dato . . . . .	19
4.2.3 Notifiche di esecuzione . . . . .	20
4.2.4 UtilityRethink . . . . .	21
4.2.5 IDbManager . . . . .	22
4.2.6 NotificationsManager . . . . .	22
4.2.7 IQueryNotification . . . . .	23
4.2.8 INotifier . . . . .	24
4.2.9 Esempio di utilizzo notificatore sulle notifiche di esecuzione	25
4.2.10 Esempio di OnNext, OnError e OnComplete . . . . .	25
4.2.11 Connection . . . . .	26
<b>5 Progettazione del server di sistema</b>	<b>28</b>
5.1 Utilizzo di docker . . . . .	28
5.2 Server a singolo nodo . . . . .	28
5.3 Server a due nodi . . . . .	29
5.4 Server a cinque nodi . . . . .	30
5.5 Conclusioni sui tre server implementati . . . . .	31
5.6 Comandi per la gestione del server di sistema . . . . .	32
<b>6 Conclusioni</b>	<b>35</b>
6.1 Risultati . . . . .	35
6.2 Sviluppi Futuri . . . . .	35

# 1 Specifiche e descrizione del problema

## 1.1 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 [17].

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 real-time* poiché mantengono i dati sul client sincronizzati con lo stato corrente del database.

L'architettura *push delle modifiche* 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 [16]. 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*.

## 1.2 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* [15]. 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.

## 1.3 Specifiche architetturali

Il sistema che è stato implementato in questo progetto coinvolge, come mostrato in figura 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.

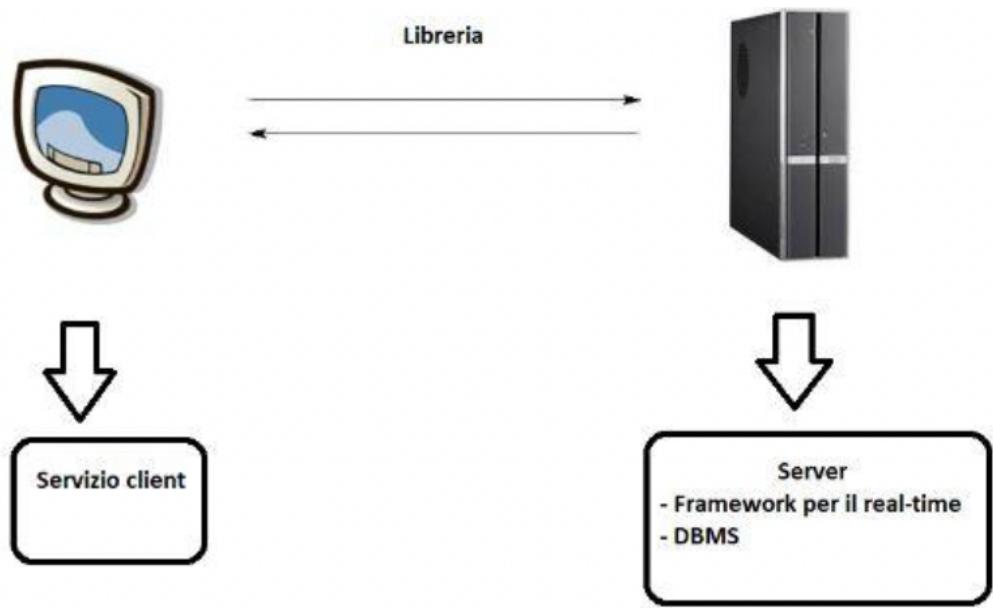


Figura 1: i tre componenti del sistema

## 1.4 Specifiche funzionali

La libreria da implementare deve consentire ai servizi client di inviare notifiche legate a eventi ad altri applicativi client interessati. Deve, per questo motivo, consentire a tutti gli applicativi di registrarsi sul server per ricevere le notifiche di interesse inviate da altri client.

Si vuole simulare un contesto nel quale i dati sono archiviati su un altro dbms (ad esempio Oracle) mentre il server *RethinkDB* sarebbe utilizzato solamente per il salvataggio delle notifiche. La libreria sarà utilizzata, ad esempio, a seguito dell'inserimento di un nuovo ordine sul dbms Oracle. L'applicativo client, oltre al nuovo dato, inserirà, tramite la libreria, una notifica con argomento uguale a "O" (Ordine) sul database del server *RethinkDB*. Saranno, di conseguenza, notificati automaticamente da *RethinkDB* tutti gli altri servizi client che si erano registrati in precedenza per la ricezione di notifiche con argomento "O". Il client che riceve la notifica potrà aggiornare il sistema di caching del nuovo evento (es. inserimento di nuovo ordine) allo scopo di fornire informazioni costantemente aggiornate agli utenti delle applicazioni web.

Oltre alla capacità di garantire la distribuzione delle notifiche, la libreria permette di eseguire operazioni CRUD sulle tabelle che sono considerate di sistema dalla libreria implementata (es. la tabella che raccoglie le notifiche chiamata *Notifications*). La libreria fornisce, infine, funzionalità gestionali per il database come la creazione e la cancellazione di nuove tabelle e di indici.

## 2 Tecnologia Scelta

Nella fase iniziale del progetto, è stato ricercato il miglior software utilizzabile per la realizzazione del sistema.

Tra tutti i software che potevano essere utilizzati per svolgere il ruolo di server del sistema, è stato scelto il dbms *RethinkDB* [12].

Si è deciso 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. Per questo ultimo motivo, analizzando le specifiche architetturali descritte in precedenza, il server di questo progetto è stato implementato utilizzando esclusivamente il dbms *RethinkDB* senza utilizzare un framework aggiuntivo che gestisca il processo di notificazione.

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 nel capitolo successivo, 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*.

### 2.1 RethinkDb

*RethinkDB* è un dbms non relazionale di tipo documentale [10] 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 essere aggiornata sui 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

*RethinkDB* è, inoltre un sistema diverso dalle API di sincronizzazione in tempo reale come *Firbase*, per tre diverse motivazioni [10]:

- 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 geospatiali 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'installazione 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 installazione leggermente più complicato 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.

## 2.2 ReQL

Per interrogare un server *RethinkDB*, come già anticipato, viene utilizzato il linguaggio *ReQL* [11]. 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 2, e le funzioni di aggregazione.

```
SELECT *
FROM posts
JOIN users
ON posts.user_id = users.id
```

```
r.table("posts").eq_join(
  "id",
  r.table("users"),
  index="id"
).zip()
```

Figura 2: esempio di query di join con linguaggio *SQL* e *ReQL*

*SQL* e *RethinkDB* condividono, inoltre, una terminologia molto simile. In figura 3 è presente una tabella dei termini e dei concetti dei due sistemi.

SQL	RethinkDB
database	database
table	table
row	document
column	field
table joins	table joins
primary key	primary key (by default <code>id</code> )
index	index

Figura 3: 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 che è stata implementata e che sarà trattata in seguito (*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.

### 3 Progettazione libreria UtilityRethink

Una delle attività principali dello studio effettuato prevede la progettazione e l’implementazione di una libreria che garantisce agli applicativi di interagire con il server di sistema. Questo capitolo, in particolare, tratta gli aspetti di progettazione della libreria, mentre l’implementazione sarà discussa nel capitolo successivo.

La tecnologia server scelta, come spiegato nel capitolo precedente, è stata il dbms non relazionale real-time chiamato *RethinkDB*.

La libreria chiamata *UtilityRethink*, che è stata sviluppata in questo progetto, oltre a garantire ai servizi client l’interfacciamento a *RethinkDB*, è stata progettata per ottenere il sistema di notifiche richiesto.

#### 3.1 Diagramma dei casi d’uso

Nello schema in figura 4 sono descritte le quattro principali funzionalità che la libreria offre ai servizi client che ne usufruiscono. In particolare garantisce:

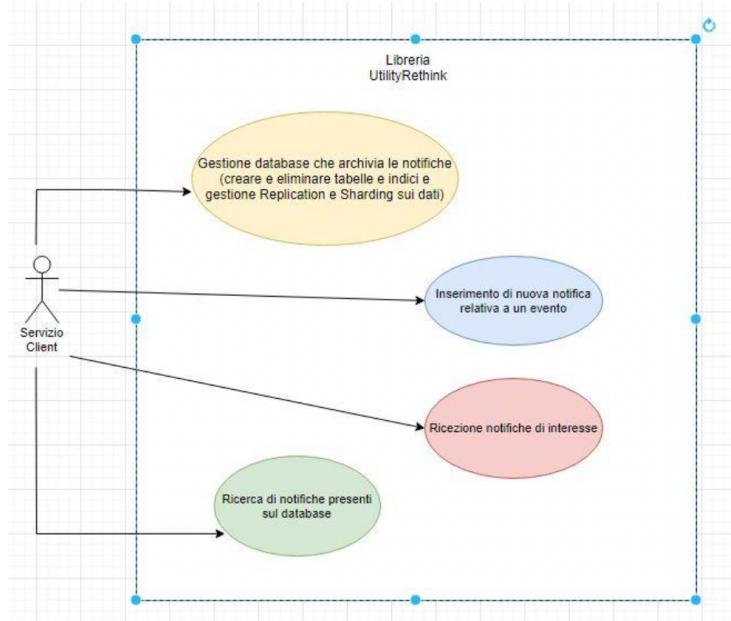


Figura 4: Diagramma dei casi d’uso della libreria *UtilityRethink*.

- la gestione del database sul server *RethinkDB* su cui sono archiviate le notifiche: creazione e cancellazione di tabelle e indici e gestione della *Replication* e *Sharding* dei dati.
- 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*;

### 3.2 Diagramma delle attività e di sequenza

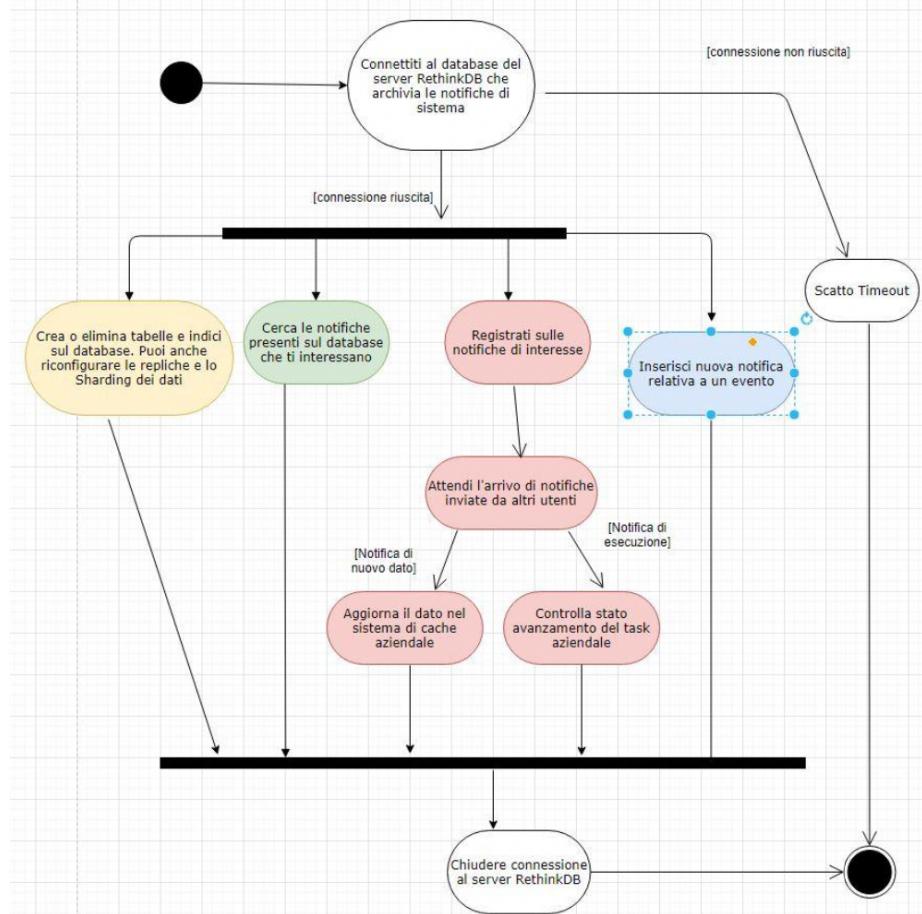


Figura 5: Diagramma delle attività che un servizio può svolgere utilizzando la libreria *UtilityRethink*.

In figura 5 sono analizzati gli scenari possibili di utilizzo della libreria *UtilityRethink* da parte di un servizio client. Ogni attività rappresentata in figura 5 è tipica di uno o tutti i casi d'uso rappresentati in figura 4. Questa connessione tra le due figure, in particolare, è denotata dai colori. Il caso d'uso Ricezione notifiche di interesse, ad esempio, che è stato colorato di rosso in figura 4, è rappresentato dalle attività colorate anch'esse di rosso in figura 5.

Le attività lasciate in bianco, invece, sono relative a tutti i casi d’uso della libreria. L’attività “Connettiti al database del server *RethinkDB* che archivia le notifiche di sistema” è comune a tutti i casi d’uso. L’utente che utilizza la libreria, infatti, deve inizialmente connettersi al server *RethinkDB* per poter interagire con il database su cui sono archiviate le notifiche di sistema, per potersi inseguito registrarsi sulle notifiche di interesse, ad esempio. 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:

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

### 3.2.1 Come sono distribuite le notifiche ai client

Il caso d’uso più interessante da analizzare, è il funzionamento della ricezione dei messaggi colorata di rosso nei diagrammi in figura 4 e 5. Quando altri utenti inseriscono sul database una nuova notifica, come mostrato nel diagramma di sequenza in figura 6, *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 tupla inserita nel database. Quando un servizio client viene “notificato” è libero di decidere come agire. Come descritto nel diagramma in figura 5 dalle attività in rosso, i servizi client utilizzeranno questo sistema di notificazione per aggiornare i dati nel sistema di cache se la notifica arrivata è di tipo nuovo dato, mentre controlleranno lo stato di avanzamento di un task/processo se la notifica arrivata è di tipo esecuzione. Il modo in cui è stato implementato il sistema di notificazione e come sono state progettate le notifiche stesse saranno trattate in seguito nel dettaglio.

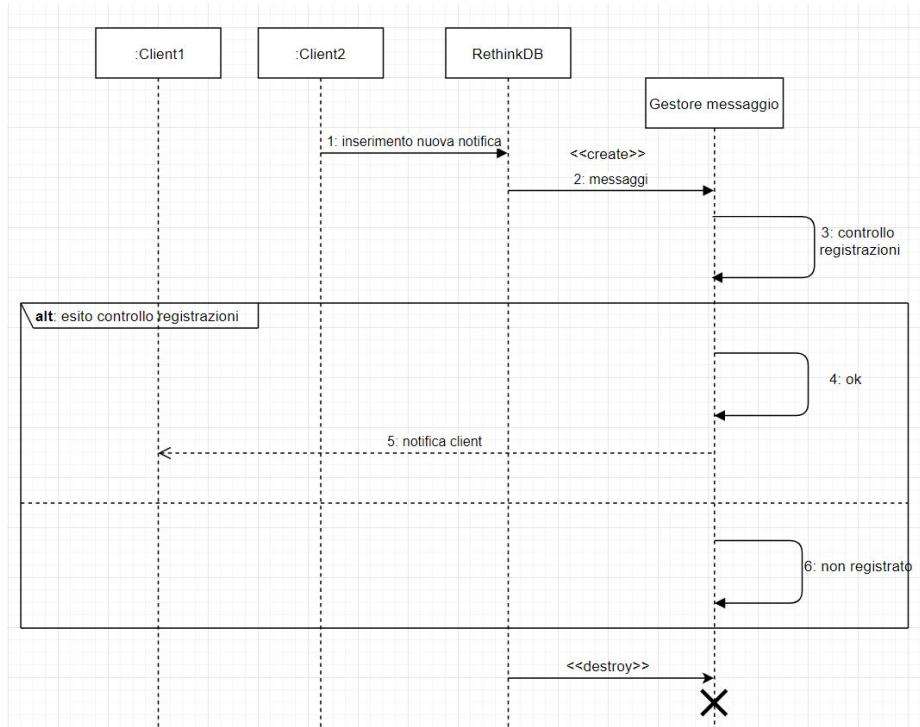


Figura 6: Diagramma di sequenza sul funzionamento della ricezione dei messaggi.

### 3.3 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. Questa tabella è stata chiamata Notifications e dovrà raccogliere e mantenere ogni tipo di notifica di sistema.

### 3.4 Progettazione notifiche

```
{  
    "Table": "Clienti" ,  
    "arg": "A" ,  
    "date": Non Dec 14 2020 15:38:58 GMT+00:00 ,  
    "id": "0c3b38f1-86a0-4a84-aaf0-098b31e7650d" ,  
    "text": "nuovo inserimento sulla tabella clienti"  
}  
  
{  
  
    "arg": "T1" ,  
    "date": Non Dec 14 2020 15:42:38 GMT+00:00 ,  
    "id": "ae5cde8e-f151-4825-9dab-daleb0078f28" ,  
    "idExec": "Task1" ,  
    "text": "Task 1 completato"  
}
```

Figura 7: 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. È stata, quindi, sfruttata questa caratteristica, tipica dei dbms non relazionali, per raccogliere i due diversi tipi di notifiche (notifiche di nuovo dato e notifiche di esecuzione) all'interno della stessa tabella *Notifications* come mostrato in figura 7. L'unico vincolo richiesto, come sarà mostrato in seguito nel capitolo dell'implementazione, è che qualsiasi nuovo tipo di notifica che sarà inserita su *Notifications*, dovrà obbligatoriamente avere i campi della classe astratta *Notification* della libreria *UtilityRethink*.

## 4 Implementazione e testing della libreria UtilityRethink

Nel capitolo precedente sono stati trattati gli aspetti di progettazione della libreria *UtilityRethink*, in questo capitolo, invece, si pone maggiormente l'accento sui dettagli implementativi. La libreria è, al momento, disponibile solamente in versione per applicativi scritti in linguaggio c. Questa versione sarà approfondita dettagliatamente in questo capitolo.

### 4.1 Software utilizzati

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)*.

*Bchavez/RethinkDb.Driver* è un driver che fornisce, agli applicativi scritti in linguaggio c, un'api utile per l'interfacciamento al server *Rethinkdb* [1]. Pur non essendo un driver “ufficiale”, è molto simile a quello scritto in java, ed è stato fondamentale per lo sviluppo della libreria.

Per il testing della libreria è stato sviluppato un progetto separato, chiamato *RethinkDbTest*, che verifica la corretta implementazione dei suoi principali componenti: *Notifier*, *QueryNotification* e *DbManager*. A tal proposito è stato utilizzato il framework per gli unit test nativo di visual studio.

Per implementare il notificatore, oltre alla funzionalità di *changefeed*, che sarà discussa in seguito, offerta dal driver *Bchavez/RethinkDb.Driver*, è stata utilizzata anche la libreria *Reactive Extensions (Rx)*. Quest'ultima è in uso, generalmente, per lo sviluppo di programmi asincroni basati su eventi utilizzando il pattern observer tipico della programmazione a oggetti e operatori in stile [9].

### 4.2 Diagrammi delle classi dei componenti di libreria

Nel diagramma delle classi, rappresentato in figura 8, sono raffigurati tutti i componenti (classi) di libreria e i legami che hanno tra loro per garantire ai servizi client le funzionalità richieste dall'applicativo.

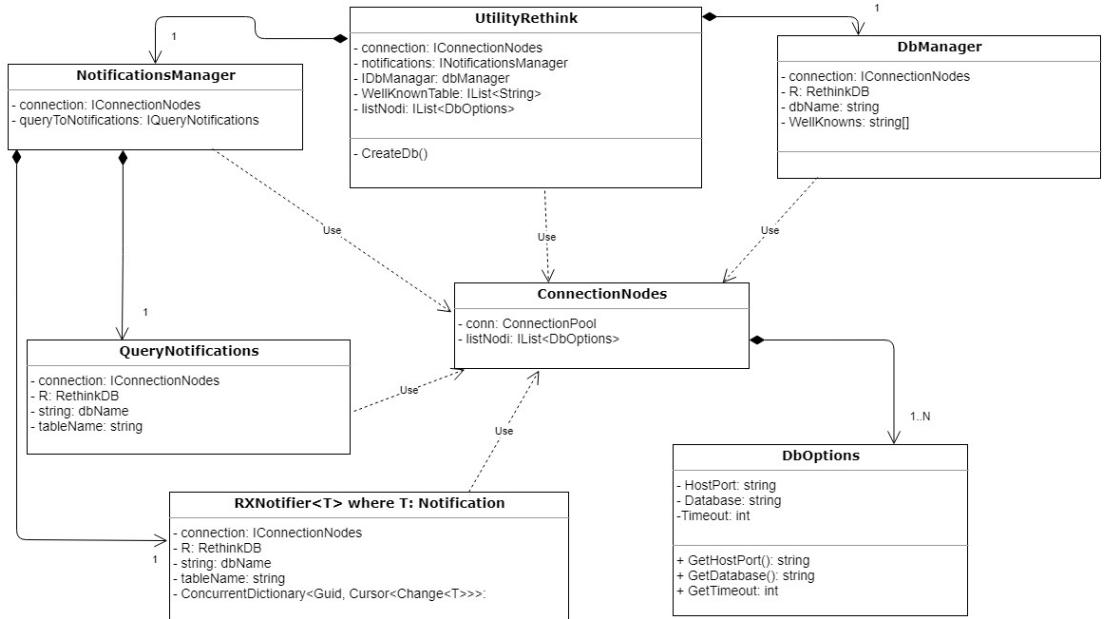


Figura 8: diagramma delle classi della libreria *UtilityRethink*.

#### 4.2.1 Notifiche

Prima di trattare il funzionamento di ogni singolo componente di *UtilityRethink*, introduciamo come sono state implementate le notifiche dal punto di vista della libreria.

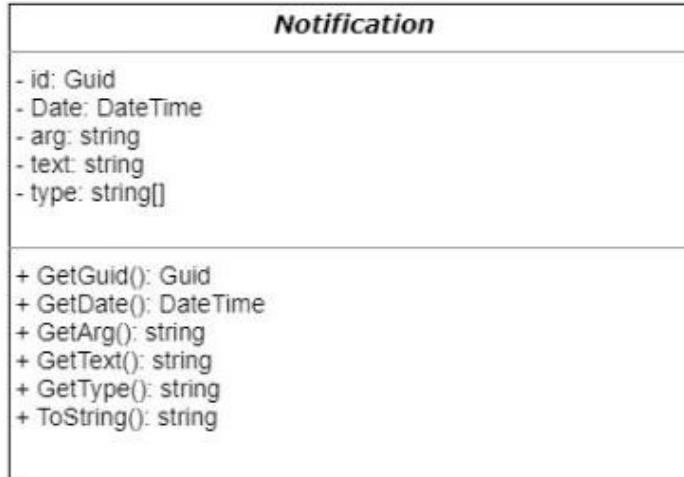


Figura 9: classe base delle notifiche *Notification*.

Ogni classe c che rappresenta un tipo di notifica deve ereditare dalla classe base *Notification* sopra elencata in figura 9. *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 nel server *RethinkDB*, non sarà un problema. Le funzionalità offerte da tutti i componenti di libreria lavorano, infatti, con qualsiasi tipo di “classe notifica” che eredita da *Notification*. La classe c che “mappera” 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*.

#### 4.2.2 Notifiche di nuovo dato

Le notifiche di nuovo dato sono una delle due notifiche richieste per il caso di utilizzo aziendale del sistema. Queste notifiche raccolgono su quale tabella del “database reale” è avvenuta un’operazione di insert, update o delete di un dato. Di conseguenza, combinando l’utilizzo di queste notifiche con il componente di libreria *INotifier* (gestore della distribuzione delle notifiche in tempo reale che sarà trattato in seguito), tutti i servizi client saranno in grado di aggiornare i dati presenti nel sistema di caching aziendale in tempo reale. 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 molto utilizzato dalle aziende 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.

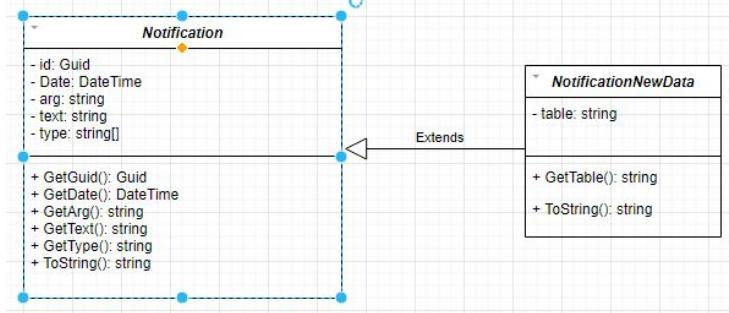


Figura 10: notifiche di nuovo dato.

Le notifiche di nuovo dato sono chiamate *NotificationNewData* dalla libreria *UtilityRethink* e, ereditando da *Notification*, come mostrato in figura 10, hanno tutti i campi base delle notifiche. Inoltre hanno un campo *Table* che serve a identificare una tabella del "database reale" su cui è avvenuta una modifica.

#### 4.2.3 Notifiche di esecuzione

Le notifiche di esecuzione sono il secondo tipo di notifica richieste per il caso d'uso del sistema. Come per le notifiche di nuovo dato, combinando queste notifiche con il componente di libreria *INotifier* i servizi client saranno aggiornati in tempo reale sullo stato di avanzamento di un Task/processo di interesse.

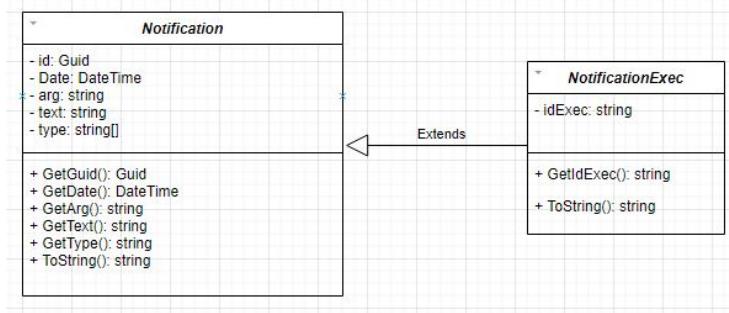


Figura 11: notifiche di esecuzione.

Le notifiche di esecuzione, come mostrato in figura 11, sono chiamate *NotificationExec* dalla libreria *UtilityRethink*. 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.

#### 4.2.4 UtilityRethink

*UtilityRethink* è la classe fondamentale della libreria che, attraverso i suoi sotto-componenti forniti, permette all'utente di interagire con il server *RethinkDB*.

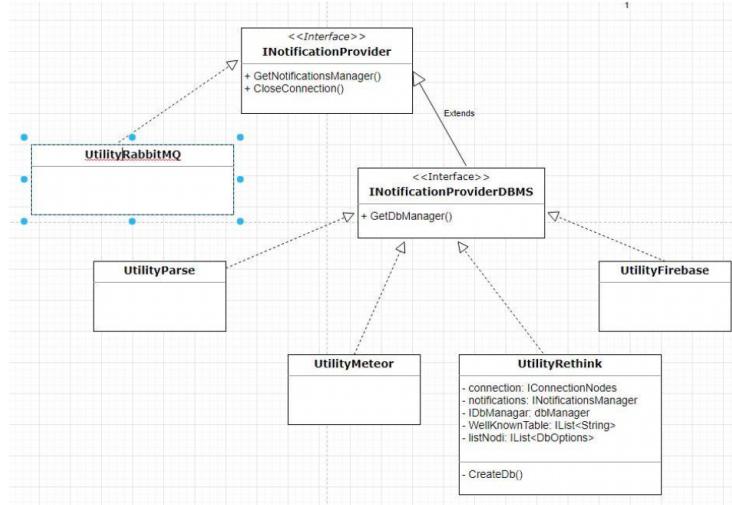


Figura 12: classe *UtilityRethink*.

I sotto-componenti forniti, come mostrato in figura 12, 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. Nell'esempio in figura 13, l'applicativo client viene fatto “connettere” a un database sul server *RethinkDB* chiamato “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 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*.

```

IList<string> hostPortsNodiCluster = new List<string>() { "192.168.1.57:28016", "192.168.1.57:28017", "192.168.1.57:28018", "192.168.1.57:28019", "192.168.1.57:28020" };
IList<string> hostPortsTwoNodi = new List<string>() { "192.168.1.57:28016", "192.168.1.57:28017" };
IList<string> hostPortsOneNode = new List<string>() { "192.168.1.57:28016" };
INotificationProviderDBMS utilityRethink = new UtilityRethink("test", hostPortsOneNode);

```

Figura 13: codice per la creazione di un’oggetto *UtilityRethink*.

#### 4.2.5 IDbManager

Il *DbManager* permette all’utente di librerie di gestire il database specificato inizialmente all’istanziamento di *UtilityRethink*.

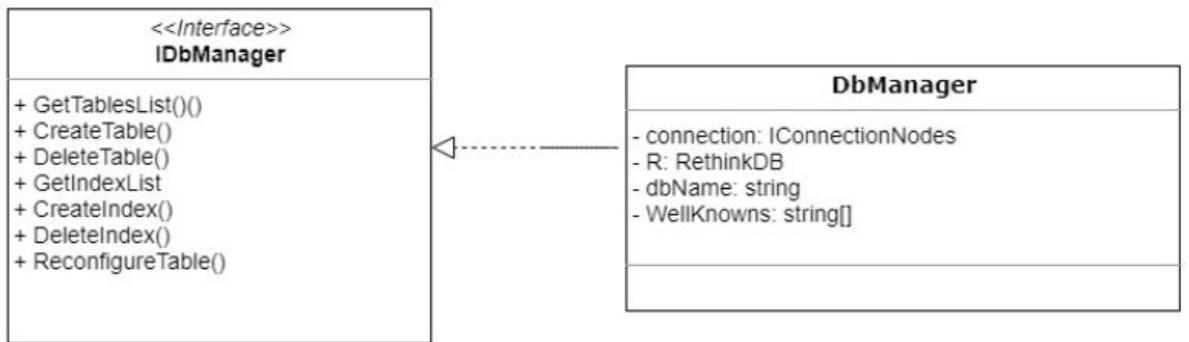


Figura 14: classe *DBManager*.

Tra le funzionalità offerte troviamo la possibilità di creare e eliminare tabelle e indici come indicato in figura 14. Il textitDBManager offre, inoltre, 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*).

#### 4.2.6 NotificationsManager

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*.

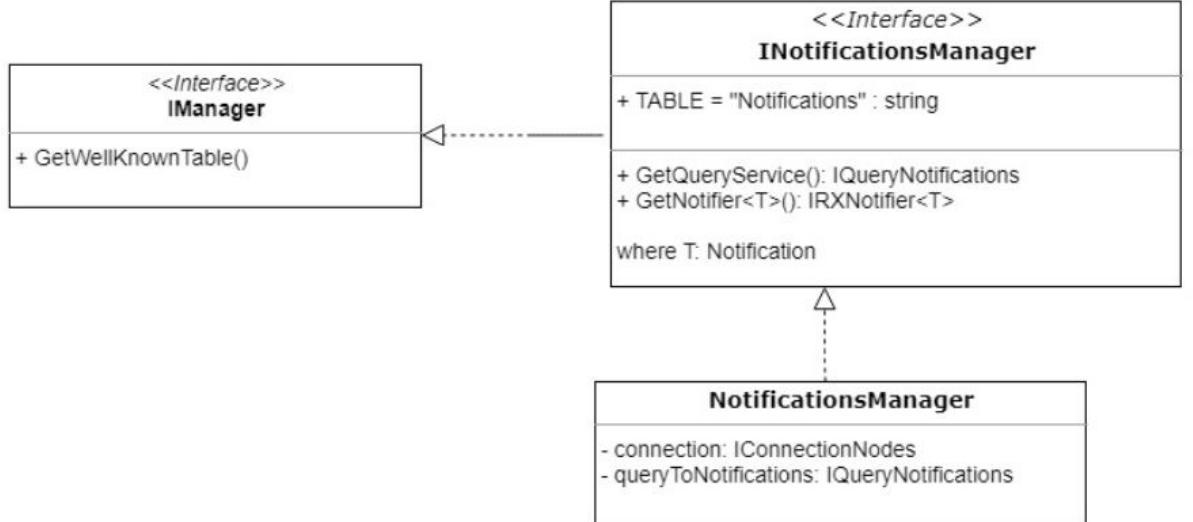


Figura 15: classe `NotificationsManager`.

`NotificationsManager` può restituire all'utente di libreria due sotto componenti fondamentali alla gestione delle notifiche del sistema: `IQueryNotification` e `IRXNotifier` (figura 15).

#### 4.2.7 `IQueryNotification`

Il `QueryNotification` è una funzione specifica offerta da `NotificationsManager` che garantisce, all'utente di libreria, la possibilità di interrogare la tabella di sistema Notifications offrendo le funzionalità di ricerca, inserimento e cancellazione di notifiche.

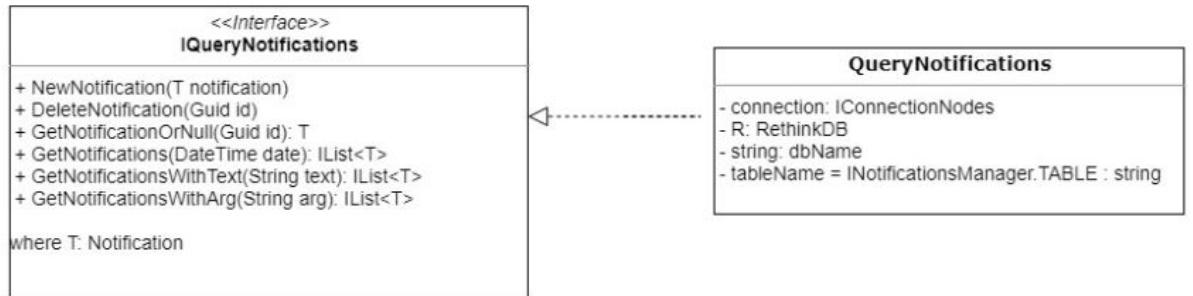


Figura 16: classe `IQueryNotifications`.

`IQueryNotifications`, come mostrato in figura 16, permette al servizio client di richiedere notifiche con determinate caratteristiche mentre per la cancellazione

occorre specificare l’“id” della notifica da eliminare. E’ un componente che gestisce qualsiasi tipo di notifica presente sulla tabella *Notifications*. Come precisato precedentemente nel capitolo riguardanti le notifiche, questa soluzione è stata implementata utilizzando le proprietà dei generici e della *Reflection* tipiche della programmazione a oggetti.

#### 4.2.8 INotifier

*INotifier* 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 scritti in linguaggio c chiamato *bchavez/RethinkDb.Driver* e della libreria *Reactive Extension (Rx)* [3].

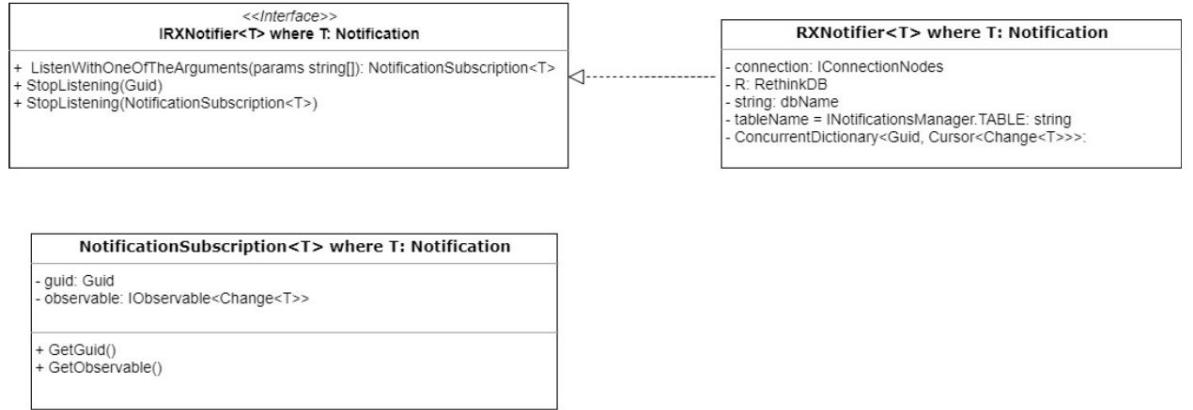


Figura 17: classe *INotifier*.

#### 4.2.9 Esempio di utilizzo notificatore sulle notifiche di esecuzione

```
NotificationSubscription<NotificationExec> pair = notifiersExec.ListenWithOneOfTheArguments("A", "B");
IObservable<Change<NotificationExec>> observableExecForArgs = pair.Observable;
observableExecForArgs.SubscribeOn(NewThreadScheduler.Default)
    .Subscribe(
        x => OnNext(x, ref onNext),
        e => OnError(e, ref onError),
        () => OnCompleted(ref onCompleted)
    );
}
```

Figura 18: esempio di utilizzo notificatore sulle notifiche di esecuzione con argomento “A” e “B”

Ad esempio, come mostrato in figura ??, 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* [2]. 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*. Ad esempio, potrebbe essere stata inserita una nuova notifica di nuovo dato sulla tabella *Notification*. Il server *RethinkDB* avviserà in automatico l’ applicativo client interessato della nuova notifica inserita. In questo modo ricevendo la notifica di nuovo dato il servizio client potrà procedere ad aggiornare il dato anche nel sistema di caching per garantire alle interfacce web di avere informazioni sempre aggiornate. In figura 18 l’applicativo client rimane in ascolto sulle 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 “*ij*” al posto di *NotificationExec* con il loro nome della classe. Anche per *INotifier*, questa soluzione è stata implementata utilizzando le proprietà dei generici e della *Reflection* tipiche della programmazione a oggetti.

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

In figura 19 troviamo un esempio di una possibile implementazione dei metodi *OnNext*, *OnError* e *OnComplete*. Ogni servizio client che utilizza la libreria deve implementare questi tre metodi per stabilire come agire per ogni notifica arrivata.

```

private static void OnCompleted(ref int onCompleted)
{
    Console.WriteLine("Stop listening");
    onCompleted++;
}

private static void OnError(Exception obj, ref int onError)
{
    Console.WriteLine("On Error");
    Console.WriteLine(obj.Message);
    onError++;
}

private static void OnNext<T>(Change<T> obj, ref int onNext) where T : Notification
{
    Console.WriteLine("On Next");
    var oldValue = obj.OldValue;

    onNext++;
    Console.WriteLine("New Value: " + obj.NewValue.ToString());
    if (oldValue != null)
    { //nel caso di un update
        Console.WriteLine("Old Value: " + oldValue.ToString());
    }
}

```

Figura 19: esempio di una possibile implementazione dei metodi *OnNext*, *OnError* e *OnComplete*.

#### 4.2.11 Connection

Il *ConnectionNodes* è la classe che rappresenta la connessione al server *RethinkDB*.

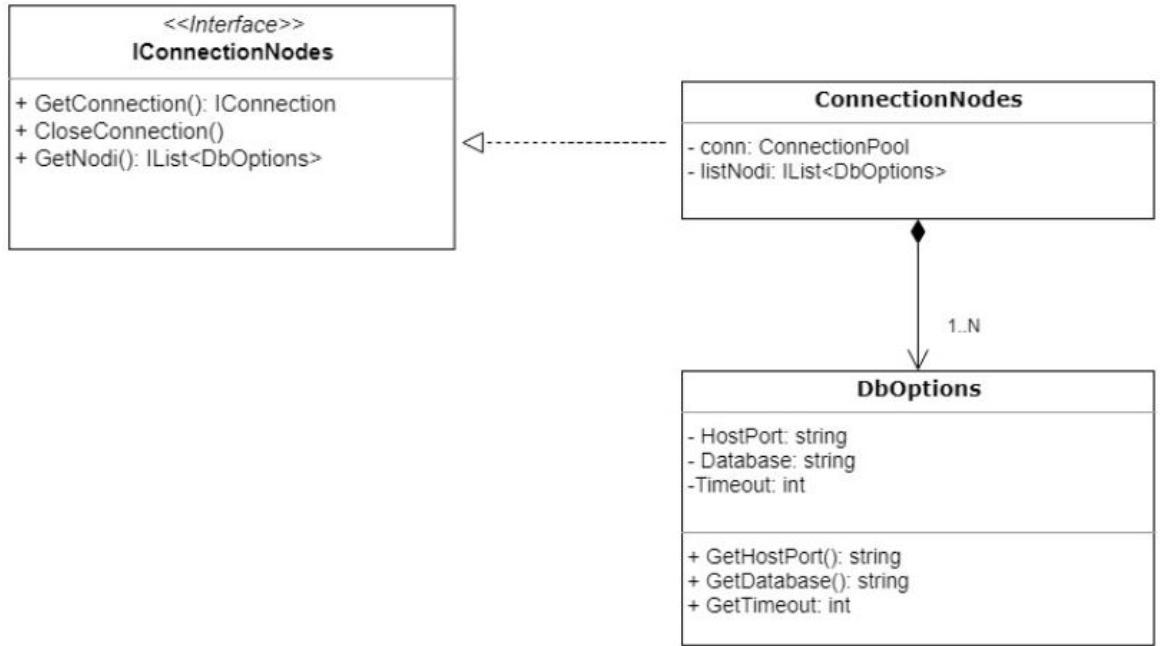


Figura 20: classe *ConnectionNodes*.

Viene istanziata, inizialmente, dalla classe principale *UtilityRethink* per poter essere utilizzata da tutti i componenti della libreria come *IQueryNotifications* e *INotifier*. I sotto-componenti di libreria, infatti, utilizzano per interfacciarsi al server la stessa istanza di *ConnectionNodes*. *ConnectionNodes* come mostrato in figura 20, 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 e 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* risulterebbe un'operazione pesante. *Conn* per connettersi ai nodi del server

## 5 Progettazione del server di sistema

Insieme alla progettazione e all'implementazione della libreria *UtilityRethink*, un'altra attività importante in questo studio è stata la progettazione del server. Tra i diversi software utilizzabili per svolgere il ruolo del server di sistema, come è già stato spiegato e discusso nel capitolo 2, è stato scelto il dbms *RethinkDB*. Per riuscire a sviluppare il server *RethinkDB* di sistema è stato fondamentale utilizzare la tecnologia offerta dai container *Docker*. Grazie ad essi, è stato possibile implementare tre diversi server *RethinkDB* (a uno o più nodi) che saranno analizzati in seguito.

### 5.1 Utilizzo di docker

*Docker Desktop* è stato il software fondamentale per la realizzazione e gestione del server *RethinkDB* di sistema [5]. *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 dei tre diversi server. 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 [7]. E' stata utilizzata la versione *Docker Desktop for windows* perché l'applicazione del progetto si esegue sul sistema operativo Windows [4]. I comandi offerti da *Docker Desktop*, come sarà analizzato nello specifico in seguito, sono stati fondamentali per l'implementazione e mantenimento dei server *Rethinkdb* containerizzato. 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* [8].

### 5.2 Server a singolo nodo

Il server *RethinkDB* a singolo nodo, come rappresentato in figura 21, è un server che è costruito utilizzando un solo container Docker *RethinkDB*. L'unico nodo disponibile di questo server sarà raggiungibile sul proprio indirizzo IP locale sulla porta 8081.

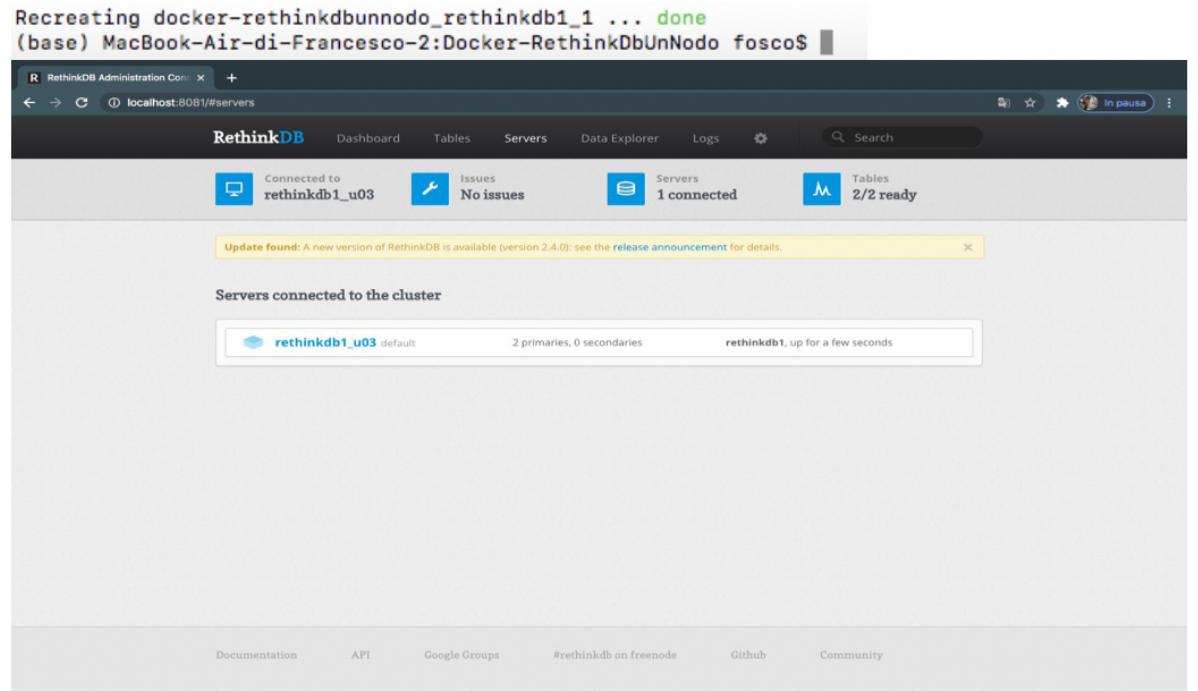


Figura 21: Server a singolo nodo.

### 5.3 Server a due nodi

Il server a due nodi, invece, come mostrato in figura 22, è formato da 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 IP locale sulle porte 8081 e 8082.

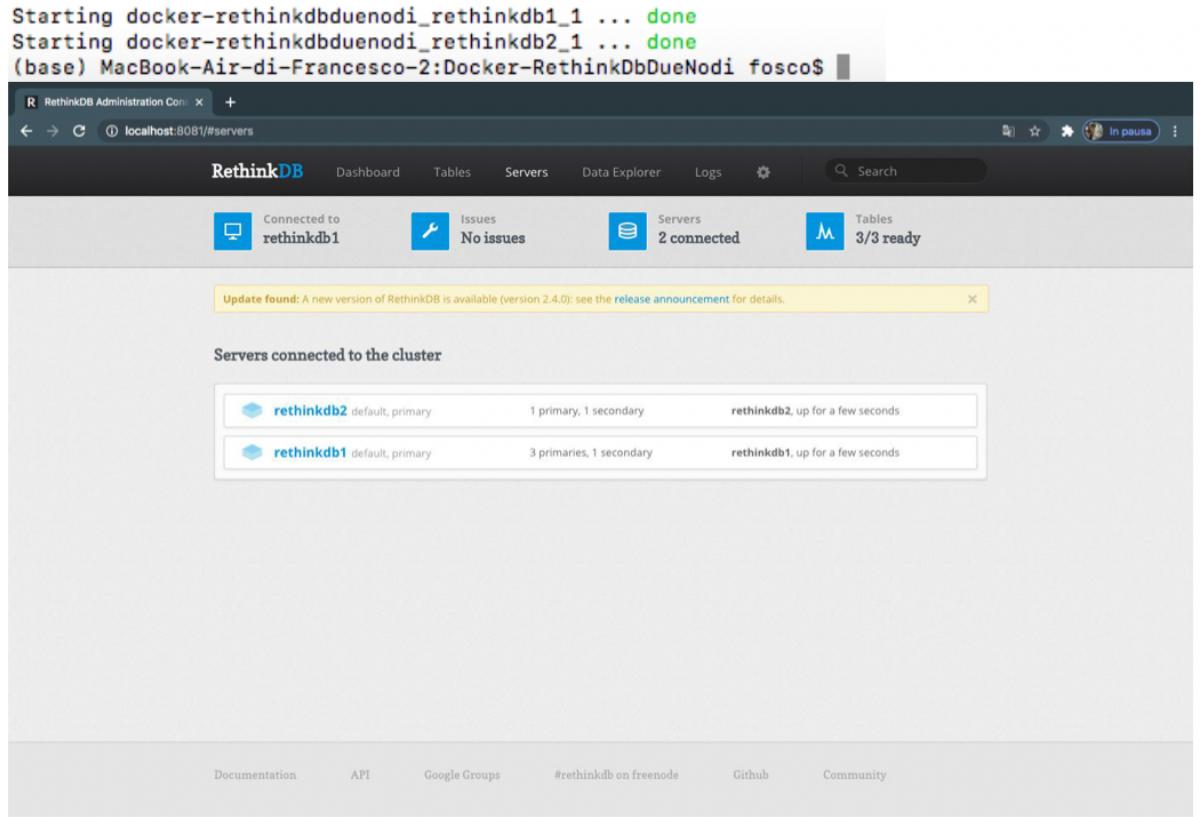


Figura 22: Server a due nodi.

#### 5.4 Server a cinque nodi

Il server a cinque nodi, infine come descritto in figura 23, 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.

```

Starting docker-rethinkdbcluster_rethinkdb3_1 ... done
Starting docker-rethinkdbcluster_rethinkdb2_1 ... done
Starting docker-rethinkdbcluster_rethinkdb4_1 ... done
Starting docker-rethinkdbcluster_rethinkdb5_1 ... done
Starting docker-rethinkdbcluster_rethinkdb1_1 ... done
(base) MacBook-Air-di-Francesco-2:Docker-RethinkDbCluster fosco$ 

```

The screenshot shows the RethinkDB Administration Console interface. At the top, there is a terminal window showing the command to start a Docker RethinkDB cluster, followed by five green 'done' messages indicating successful startup. Below the terminal is the web-based administration interface.

The interface has a header bar with tabs for Dashboard, Tables, Servers, Data Explorer, Logs, and a search bar. Below the header, there are four status indicators: 'Connected to rethinkdb1' (blue icon), 'Issues No issues' (green icon), 'Servers 5 connected' (blue icon), and 'Tables 2/2 ready' (blue icon).

A yellow banner at the top of the main content area states: "Update found: A new version of RethinkDB is available (version 2.4.0); see the [release announcement](#) for details." Below this, a section titled "Servers connected to the cluster" lists five servers:

Server	Type	Configuration	Status
rethinkdb2	default, primary	1 primary, 0 secondaries	rethinkdb2, up for 3 minutes
rethinkdb3	default, primary	1 primary, 0 secondaries	rethinkdb3, up for 3 minutes
rethinkdb5	default, primary	0 primaries, 2 secondaries	rethinkdb5, up for 3 minutes
rethinkdb4	default, primary	0 primaries, 1 secondary	rethinkdb4, up for 3 minutes
rethinkdb1	default, primary	1 primary, 1 secondary	rethinkdb1, up for 3 minutes

At the bottom of the interface, there are links to Documentation, API, Google Groups, #rethinkdb on freenode, Github, and Community.

Figura 23: Server a cinque nodi.

## 5.5 Conclusioni sui tre server implementati

Ogni *container* che è stato sviluppato rappresenta di fatto un nodo di uno dei tre server. Nei due casi in cui il server è formato da più nodi, è stato utilizzato un particolare “sistema di tagging” tipico di *RethinkDB* [13]. Questi tag sono definiti dall’utente e servono a identificare i nodi. A ogni nodo in un cluster *RethinkDB*, come indicato in figura 24 è possibile assegnare zero o più tag.

```
rethinkdb --server-tag us --server-tag us_west
```

Figura 24: tag dei nodi *RethinkDB*.

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 eseguiti i comandi necessari per il join dei nodi in un unico server [14]. 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* [6]. 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.

## 5.6 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 25, 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.

```

(base) AirdiFrancesco2:Docker-RethinkDbDueNodi fosco$ docker-compose -f docker-compose.yml build
Building rethinkdb1
Step 1/5 : FROM rethinkdb:2.3.5
--> be24926bde9a
Step 2/5 : COPY ./start-rethinkdb-cluster.sh /data
--> Using cache
--> d9d449308bea
Step 3/5 : RUN chmod 777 /data/start-rethinkdb-cluster.sh
--> Using cache
--> cf998013a64b
Step 4/5 : RUN ls -al /data
--> Using cache
--> d9355c8eac0
Step 5/5 : CMD ["bash", "-c", "/data/start-rethinkdb-cluster.sh"]
--> Using cache
--> 456fba3cc875
Successfully built 456fba3cc875
Successfully tagged docker-rethinkdbduenodi_rethinkdb1:latest
Building rethinkdb2
Step 1/5 : FROM rethinkdb:2.3.5
--> be24926bde9a
Step 2/5 : COPY ./start-rethinkdb-cluster.sh /data
--> Using cache
--> d9d449308bea
Step 3/5 : RUN chmod 777 /data/start-rethinkdb-cluster.sh
--> Using cache
--> cf998013a64b
Step 4/5 : RUN ls -al /data
--> Using cache
--> d9355c8eac0
Step 5/5 : CMD ["bash", "-c", "/data/start-rethinkdb-cluster.sh"]
--> Using cache
--> 456fba3cc875
Successfully built 456fba3cc875
Successfully tagged docker-rethinkdbduenodi_rethinkdb2:latest
(base) AirdiFrancesco2:Docker-RethinkDbDueNodi fosco$ █

```

Figura 25: 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 26, è ora online e i suoi nodi sono in attesa di rispondere alle richieste delle applicazioni client.

```

(base) AirdiFrancesco2:Docker-RethinkDbDueNodi fosco$ docker-compose -f docker-compose.yml up -d
Creating network "docker-rethinkdbduenodi_rethink-net" with driver "bridge"
Creating docker-rethinkdbduenodi_rethinkdb2_1 ... done
Creating docker-rethinkdbduenodi_rethinkdb1_1 ... done
(base) AirdiFrancesco2:Docker-RethinkDbDueNodi fosco$ █

```

Figura 26: up del server RethinkDB a due nodi

Infine, come mostrato in figura 27, 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.

```
(base) AirdiFrancesco2:Docker-RethinkDbDueNodi fosco$ docker-compose -f docker-compose.yml stop  
(base) AirdiFrancesco2:Docker-RethinkDbDueNodi fosco$ █
```

Figura 27: stop del server *RethinkDB* a due nodi.

## 6 Conclusioni

### 6.1 Risultati

L’obiettivo di questo progetto di tesi è stato lo sviluppo di un sistema che garantisca la distribuzione di notifiche tra vari servizi software. Inizialmente è stato analizzato il contesto in cui collocare il presente elaborato, descrivendo i nuovi sistemi *database real-time* e, successivamente, si è proceduto a un’attenta fase di analisi del sistema descrivendo i principali requisiti dell’applicazione e scegliendo le tecnologie da utilizzare. Per la progettazione del sistema sono state svolte due attività principali: lo sviluppo del server e la progettazione e implementazione di una libreria che garantirà ai servizi client l’interfacciamento con il server stesso. I tre server sviluppati (a uno, due e cinque nodi) sono dei prototipi iniziali che saranno sicuramente modificati e migliorati in futuro, mentre la libreria *UtilityRethink* soddisfa tutti i principali requisiti iniziali. Oltre a garantire all’utente la possibilità di inviare e ricevere nuove notifiche relative a eventi, la libreria garantisce anche il salvataggio delle notifiche al fine di revisionarle in un secondo momento.

### 6.2 Sviluppi Futuri

L’applicazione di oggi è migliorabile in futuro. Per rendere la libreria ancora meno dipendente dalla tecnologia scelta per il server è stata implementata l’interfaccia *INotificationProvider* come mostrato in figura 28. In questo modo, nel caso in cui si volesse sostituire *RethinkDB* con un’altra tecnologia (es. *RabbitMQ*) sarà sufficiente implementare la sua classe di ”utility”.

Una modifica futura sarà sicuramente effettuata sul server del sistema. In questa applicazione, come è già stato specificato, le tre versioni del server (a uno, due, cinque nodi) risiedono interamente sulla macchina locale, perché sono stati utilizzati i comandi di Docker-compose. Si potrà quindi intervenire in futuro applicando al sistema tecnologie per distribuire i diversi nodi del server su più macchine fisiche. *Docker Swarm* e *Kubernetes* sarebbero, ad esempio, due tecnologie di riferimento. In seguito, si potrà procedere allo studio della *Replication* e *Sharding* applicata da *RethinkDB* sui propri dati.

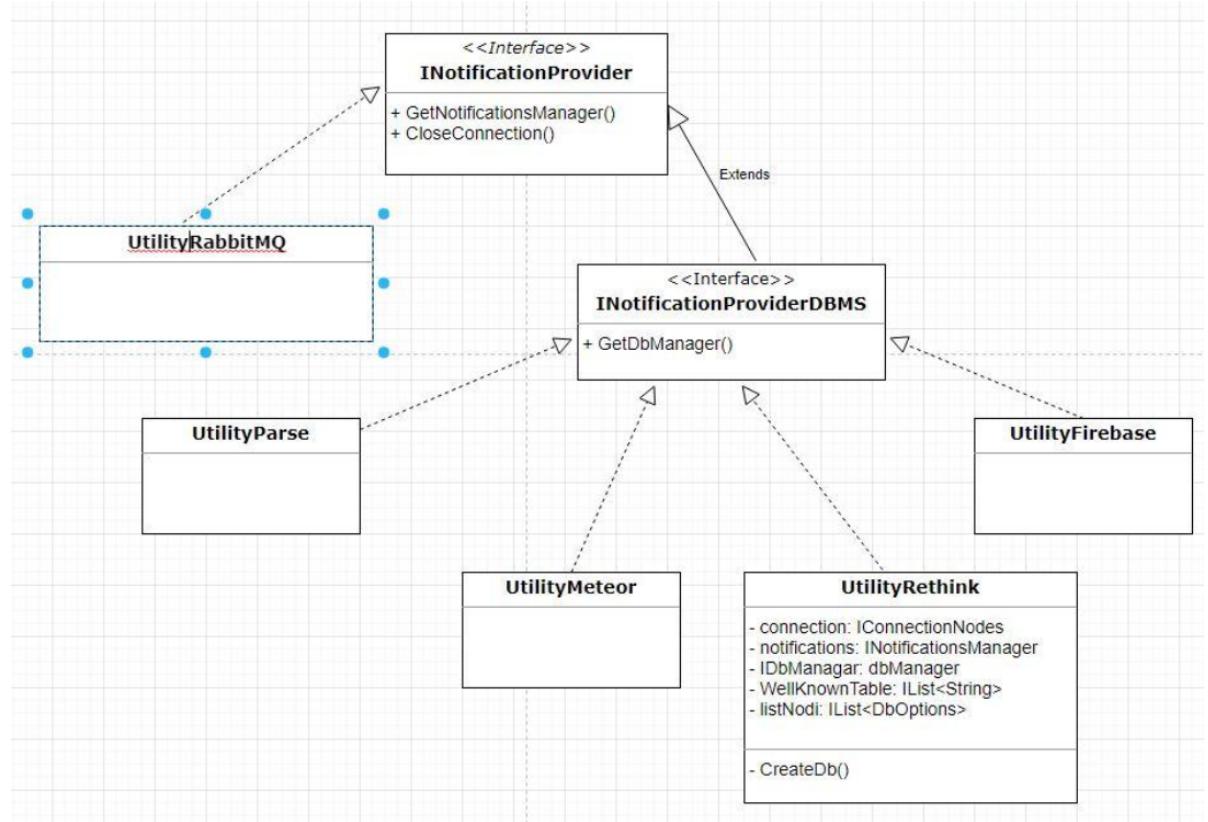


Figura 28: Classe `INotificationProvider`.

## Riferimenti bibliografici

- [1] Brian Chavez. bchavez/rethinkdb.driver. Available on line.
- [2] Brian Chavez. Esempio di utilizzo combinato delle librerie bchavez/rethinkdb.driver e reactive extension. Available on line.
- [3] Brian Chavez. Extra c driver features. Available on line.
- [4] Docker. Docker desktop for windows user manual. Available on line.
- [5] Docker. Docker desktop install docker desktop - the fastest way to containerize applications. Available on line.
- [6] Docker. Overview of docker compose. Available on line.
- [7] Docker. Use containers to build, share and run your applications. Available on line.
- [8] Docker Hub. Rethinkdb official image. Available on line.
- [9] ReactiveX. Reactivex. an api for asynchronous programming with observable streams. Available on line.
- [10] Rethinkdb. Frequently asked questions. Available on line.
- [11] Rethinkdb. Introduction to rql. Available on line.
- [12] Rethinkdb. Rethinkdb, the open source database for the realtime web. Available on line.
- [13] Rethinkdb. Server tags. Available on line.
- [14] Rethinkdb. Start a rethinkdb server. Available on line.
- [15] Wikipedia. Publish/subscribe. Available on line.
- [16] Wikipedia. Real-time database. Available on line.
- [17] Wolfram Wingerath. A real-time database survey: The architecture of meteor, rethinkdb, parse firebase, 2017. Available on line.