UNIVERSITÀ DEL PIEMONTE ORIENTALE

Relazione sul Progetto di Reti 2 – Sperimentazioni

Sperimentazione di un plugin per RabbitMQ: Gestione Abbonamenti Centro Sportivo

10025213 – Buratto Gabriele

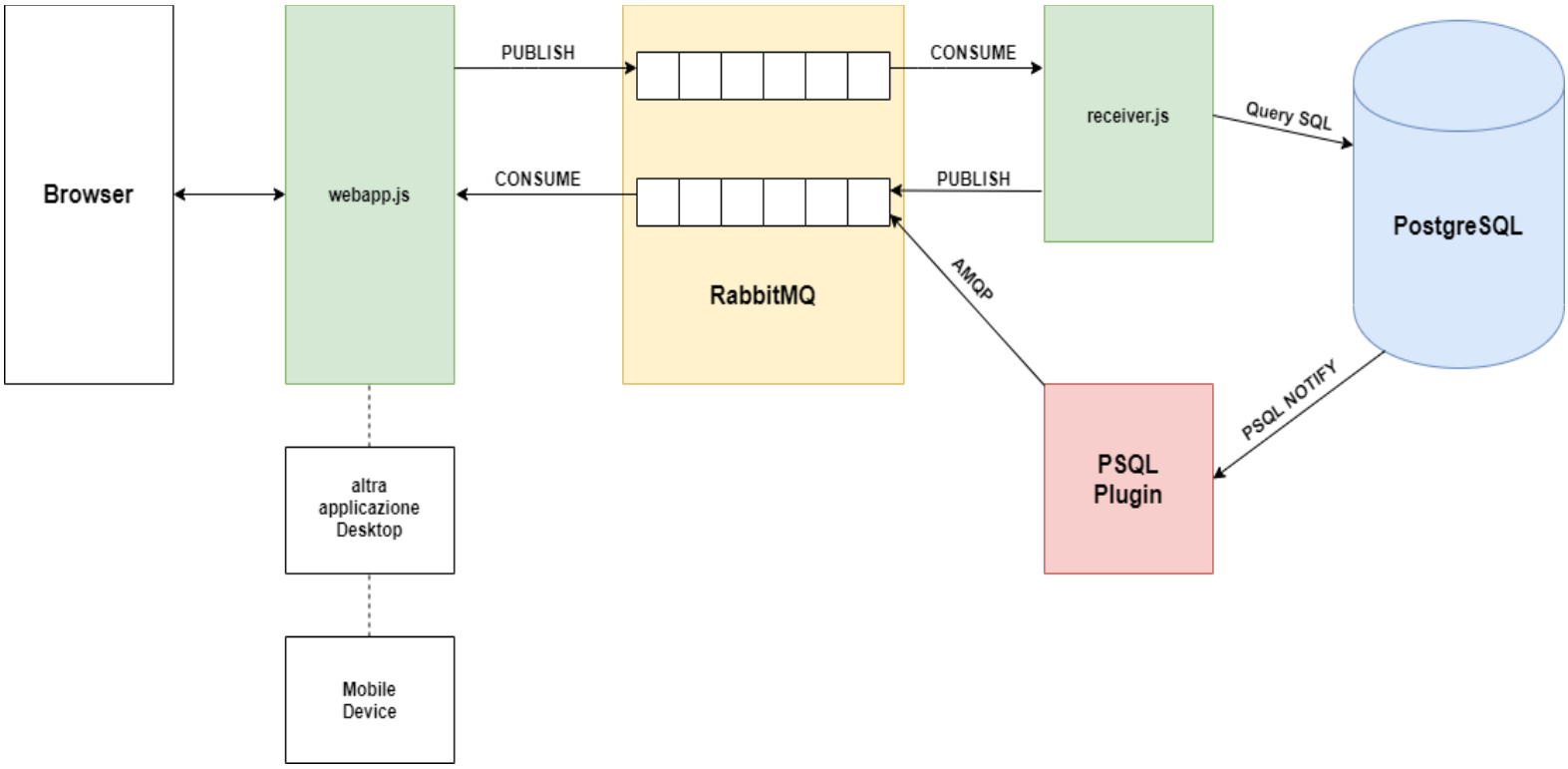
Introduzione

Lo scopo di questa relazione è stato quello di sperimentare il funzionamento di un plugin per RabbitMQ il quale si interfaccia tramite database PostgreSQL. Si è lavorato sull'integrazione della gestione degli abbonati di un centro sportivo, in particolare sulla registrazione di un utente, mostrando in tempo reale alcune operazioni svolte sul database rendendo possibile visualizzare le notifiche tramite un qualsiasi dispositivo in grado di collegarsi ad una coda RabbitMQ.

L'applicazione è in grado di comunicare con la coda sia in invio che in ricezione. Nella fase di invio, i dati ottenuti tramite la registrazione di un utente nel form (nuovo iscritto) vengono codificati in un messaggio e pubblicati nella coda. Successivamente, si è creato un worker che è sempre in ascolto sulla stessa coda, il quale è in grado di ricevere i dati e connettersi al database effettuando delle query SQL. A questo punto interviene il plugin oggetto dello studio[1] per RabbitMQ che, connesso al database, è in grado di trasformare delle notifiche PostgreSQL in messaggi AMQP che vengono pubblicati in una coda di risposta.

In questo modo si possono ricevere aggiornamenti su qualsiasi operazione effettuata sul database, anche svolte da altre applicazioni. La risposta viene letta dall'applicazione in uso, la quale è in ascolto sulla coda delle risposte venendo visualizzata in una pagina web in tempo reale.

Schema Generico



Che cos'è un message broker?

Un broker di messaggi(in questo caso, è stato usato **RabbitMQ**) è un elemento di telecomunicazione tra applicazioni software, dove queste comunicano attraverso veri e propri scambi di messaggi. Lo scopo principale di un broker è di prendere i messaggi in arrivo da queste applicazioni ed eseguire azioni su di esse. Ad esempio, un broker di messaggi può essere utilizzato per gestire una coda di carico di lavoro o una coda di messaggi per più ricevitori, fornendo memoria affidabile, con una consegna di messaggi garantita. Nel tipico scenario dello scambio di messaggi, RabbitMQ introduce, oltre alla presenza del **Publisher**, del **Consumer** e della **Queue**, un altro elemento: l'**Exchange**. Attraverso questa serie avviene l'implementazione del protocollo **AMQP**.

Tool e Software Utilizzati

Le applicazioni implementate sono state scritte con linguaggio JavaScript. Per quanto riguarda la pagina web riguardante il form di iscrizione di un utente, è stato usato HTML con uso del framework Bootstrap[2].

Per quanto riguarda la parte scritta in JavaScript è stato usato il framework Node.js[3] che, a sua volta usa il framework Express.js[4].

Node.js è utilizzato particolarmente per la gestione di applicazioni lato server senza subire blocchi nei processi. Questo permette di lavorare meglio su sistemi scalabili.

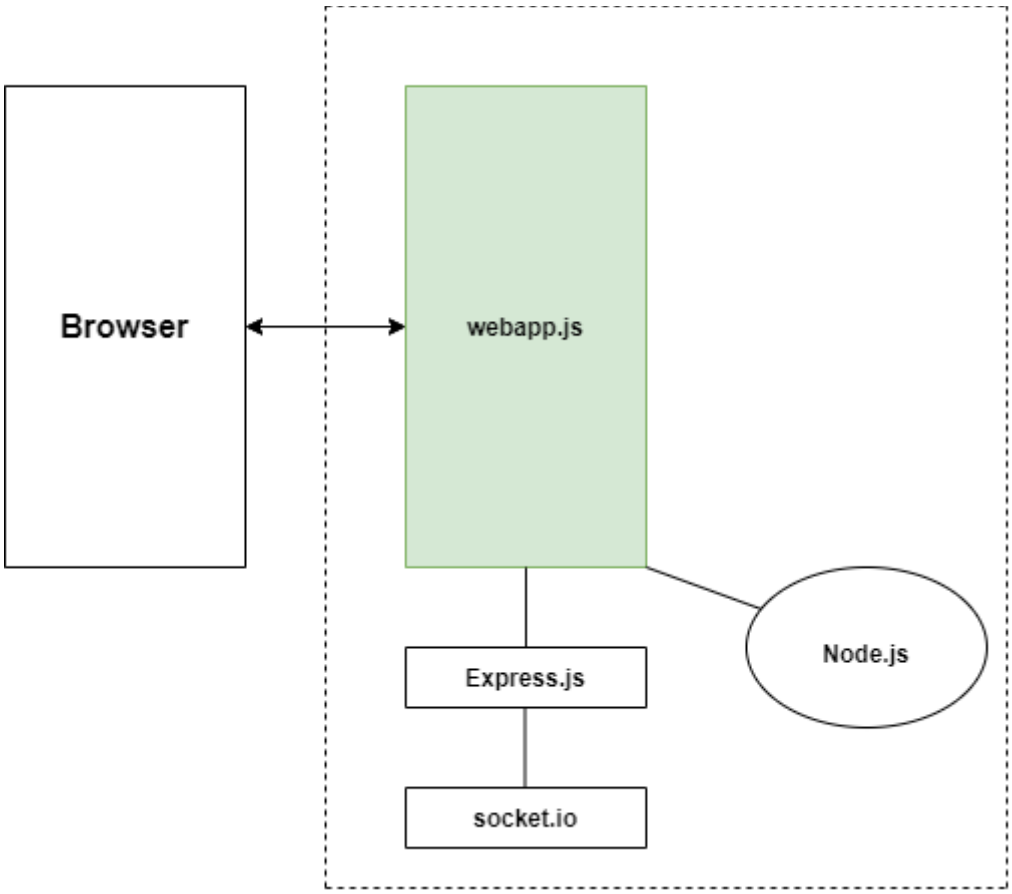
Express.js è un framework per Node.js leggero, che fornisce molte funzioni avanzate per molte applicazioni web e dispositivi mobili.

Il sistema è stato configurato localmente, servendosi di una macchina virtuale (VM, Virtual Machine) Linux usando il software Docker Toolbox[5]. Esso permette l'utilizzo di Docker in questo caso su sistema Windows, per poter eseguire dei container con sistema operativo Linux. In pratica si avvale di una Virtual Machine in esecuzione sul sistema VirtualBox con al suo interno il demone di Docker. E' stata definita un'immagine Docker(da dockerfile) per la configurazione e l'esecuzione di un broker RabbitMQ 3.5[6].
N.B. L'uso del plugin che è stato sperimentato è compatibile fino alla versione indicata.

Inoltre, tramite il package manager NPM per Node.js è stato possibile poter installare il software necessario per la corretta esecuzione dell'applicazione. Per la gestione del database si è lavorato tramite la shell psql e la versione di pgAdmin 4[7], in particolare per la visione delle tabelle e tuple, oltre alla definizione principale dei tipi di dati utilizzati.

E' stato importante infine servirsi delle librerie socket.io[8], pg[9] e amqplib[10][11]. Tutte queste sono state importate nel codice delle applicazioni Node.js. La prima (socket.io) è una libreria utile per ricevere e inviare messaggi al/dal frontend; la seconda (pg) è una collezione di moduli utili ad interfacciarsi al database PostgreSQL; la terza (amqplib) è una libreria che implementa il protocollo AMPQ necessario per poter utilizzare le API di RabbitMQ da Node.js.

Analisi della tecnologia utilizzata



L'applicazione webapp.js comprende le librerie ed i framework indicati dalla figura. Nelle righe del codice si notano questi riferimenti, ad esempio per Express.js:

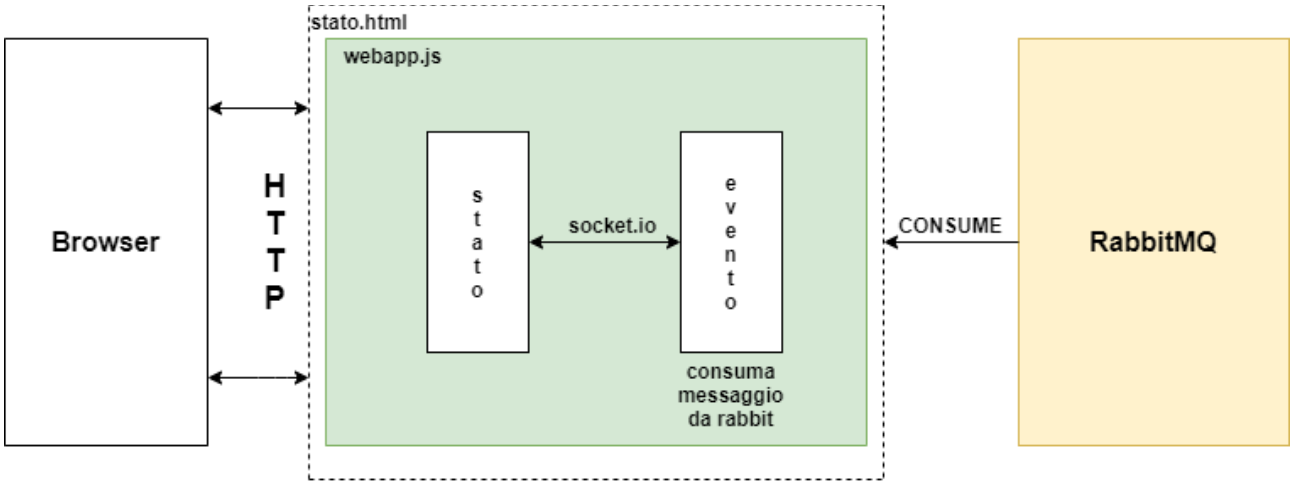
const express = require('express');
const app = express();

L'inclusione della libreria amqplib:

// queue message protocol
const amqp = require('amqplib');

Nello specifico, per quanto riguarda la libreria *socket.io*, come indicato nel codice sottostante:

// real-time communication agent
var io = require('socket.io')(http);



Questa libreria è stata usata principalmente per usare veri e propri “messaggi di stato”, inviati o ricevuti tramite il frontend utilizzato. La pagina *stato.html* fa si che si possano vedere in tempo reale le operazioni che verranno effettuate sul database. Nella pagina *webapp.js* si possono notare porzioni di codice che potranno interagire con la pagina *stato.html*:

//endpoint che serve la pagina di stato
app.get('/stato', function (req, res) {
* res.sendFile(__dirname + '/html/**stato.html**');*
});

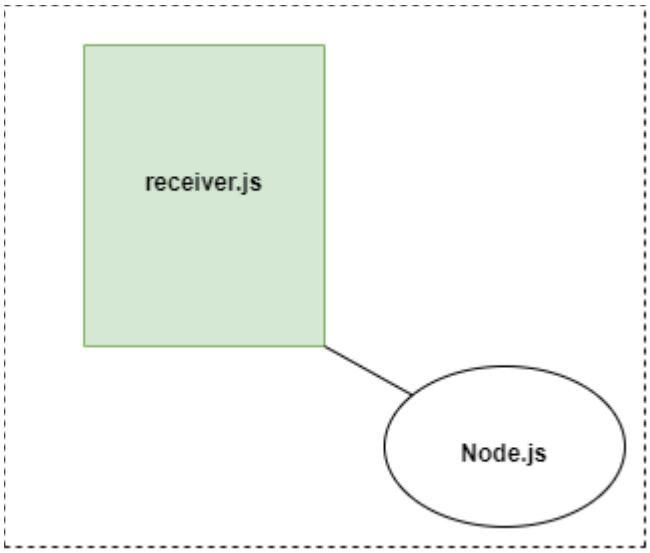
Si nota che nella creazione di questa applicazione(app) si controllano i percorsi, contrassegnati da un URL al quale l'applicazione stessa deve saper fornire una risposta. Questa gestione dei percorsi è detta routing.
E' sufficiente inserire i percorsi (diversi URL, contrassegnati dal carattere '\') ai quali l’ applicazione deve rispondere. Una funzione di callback viene richiamata quando qualcuno chiama questo percorso.
In particolare queste notazioni sono state utilizzate per quanto riguarda le risposte date in relazione al form e possono essere inviate anche ad altre pagine tramite operazioni *POST*, *GET* o *PUT*.

```
// Gestione richiesta form di iscrizione, riceve dei dati ed esegue un'azione
app.post('/api/v1/processData', async function (req, res) {
…..
…..
…..
…..
io.emit('richieste', { requestid: requestId, result: "PENDING" });

//creo endpoint per put, prendo l'id utente da cancellare dal db, accetta http put
app.put('/cancella/:id', async function(req,res){
…..
…..
…..
…..
io.emit('richieste', { requestid: requestId, result: "PENDING" });
```

Come evidenziato in grassetto, cosa fa la *emit*? E' molto importante poiché tramite questa si invia un messaggio all'oggetto client con il quale si sta parlando in modo diretto. Questo client è rappresentato dal percorso '*richieste*' riferito alla pagina *stato.html* che contiene gli oggetti che verranno modificati in tempo reale i quali sono in ascolto dalla pagina *webapp.js*.

In *stato.html*, verranno aggiornati i parametri evidenziati sotto in grassetto. Il comando *socket.on* fa si che avvenga correttamente l'ascolto tra client e server:

```
socket.on('richieste', function (data) {
  console.log("richiesta processata", data);
  /*if (data.hasOwnProperty('op') && data.op == 'DELETE')
  {
        console.log("elimino la linea di id", data.id);
  }*/
  //console.log("data",data.dataiscrizione);
  $("#richiesta").html("<p>"+data.requestid+"</p>");
  $("#risultato").html("<p>"+data.result+"</p>");

});
```



La pagina *receiver.js* è dedicata al consumo del messaggio attraverso RabbitMQ ed in particolare si connette al DB PostgreSQL, eseguendo anche le queries dedicate allo scopo:

```
//configuro postgres
const { Client, Pool} = require('pg')

const client = new Client({
user: 'postgres',
host: 'localhost',
database: 'test',
password: 'admin',
port: 5432,
})

client.connect()

const amqp = require('amqplib');
```

Nello specifico vengono incluse le librerie per la connessione a RabbitMQ(libreria amqlib) ed a PostgreSQL elencando username, password, hostname, nome del DB sul quale si lavora e porta dedicata; dopodiché viene eseguita la connessione tramite la *client.connect()*.

Si nota poi a quale indirizzo IP fare riferimento per l'ascolto e la trasmissione di messaggi tramite RabbitMQ. L'indirizzo fa riferimento all'indirizzo IP della VM, tramite il protocollo amqp[12], che è il protocollo standard per la messaggistica:

// RabbitMQ connection string
const messageQueueConnectionString = 'amqp://192.168.99.100/';

Quindi, i messaggi verranno ascoltati e consumati dal canale dedicato.

Per l'esecuzione delle queries sono state svolte funzioni dedicate, in particolare:

function inserisci_db(data){
…
…
}

function seleziona_db(data){
…
…
}

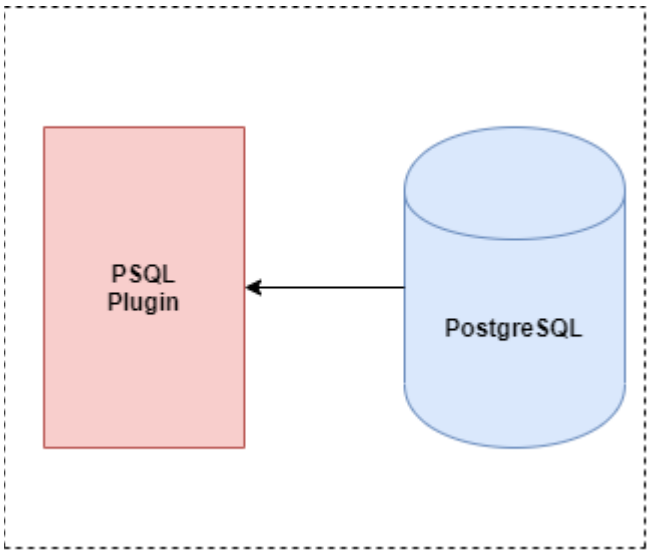
function elimina_db(data){
…
…
}

All'interno di queste funzioni vengono eseguite le opportune modifiche per quanto riguarda il lavoro all'interno del DB PostgreSQL. Sono contenuti i campi delle tabelle e le principali operazioni da fare all'interno del DB stesso.

Il vero e proprio lavoro viene svolto all'interno della funzione consume:

// consume messages from RabbitMQ
function consume({ connection, channel, resultsChannel }) {
….
….
….
}

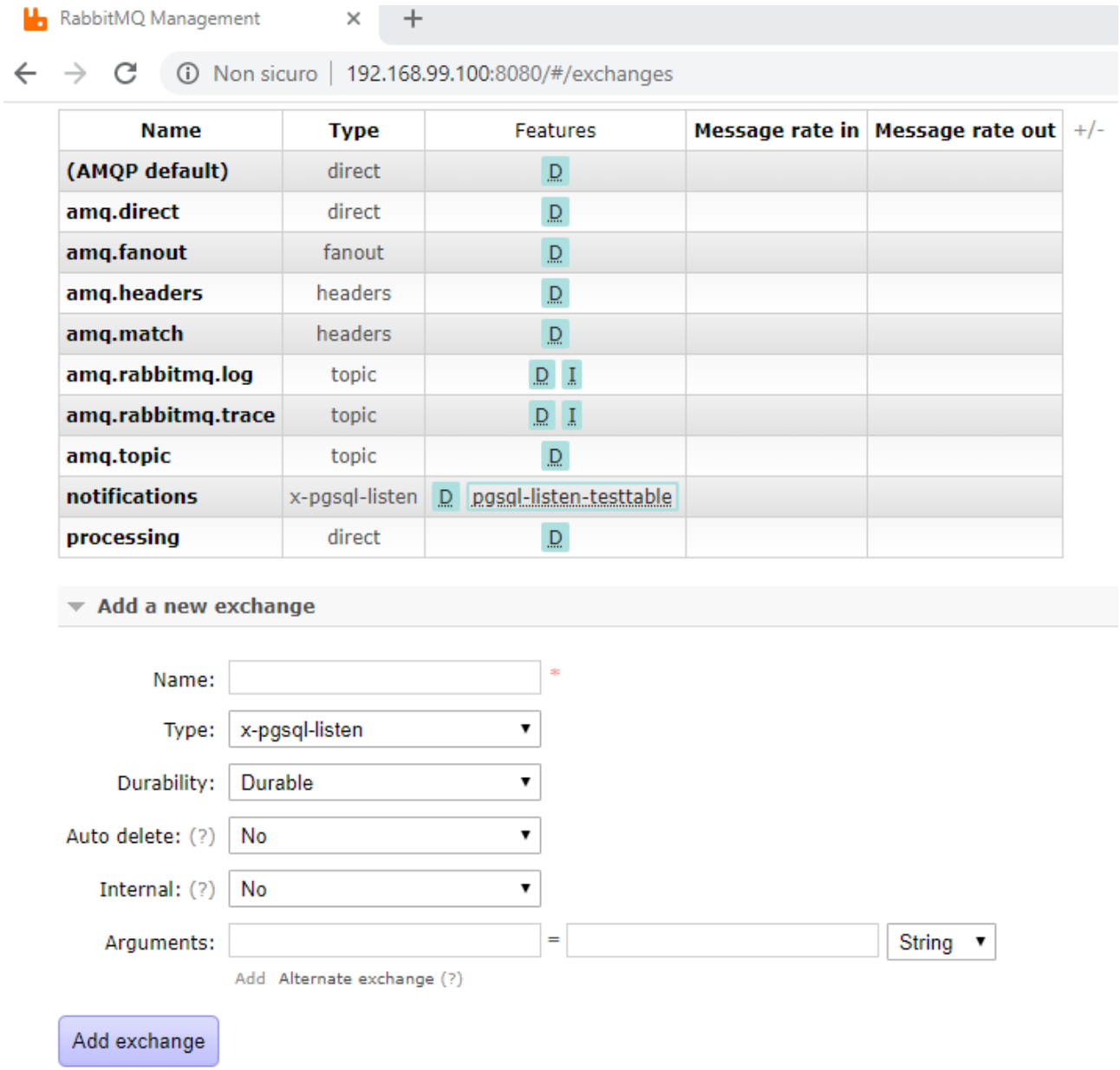
In questo modo, in base alla coda delle richieste di RabbitMQ i messaggi verranno consumati.
Alla particolare richiesta svolta sul DB (SELECT, DELETE o INSERT) si riesce ad ottenere il risultato della query voluta. Le tabelle del DB verranno aggiornate e si può ottenere il controllo in tempo reale di queste operazioni svolte, riuscendo a ricevere il messaggio di risposta sulla coda dei risultati di RabbitMQ.



Grazie al plugin PSQL vi è il collegamento tra il database e RabbitMQ, sul quale poter ottenere oppure pubblicare direttamente i messaggi. Ciò è possibile attraverso due fasi:

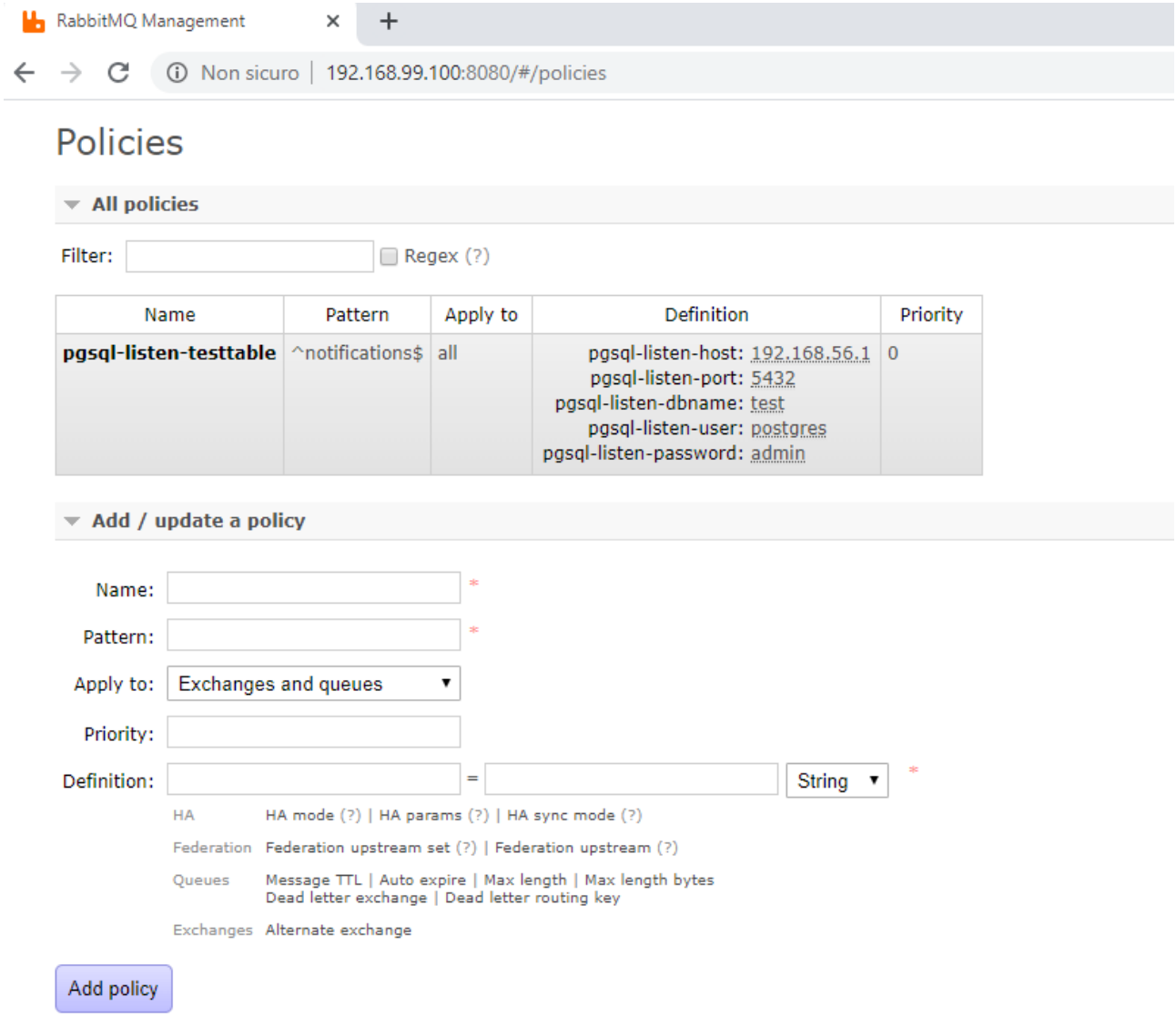
- la pubblicazione di messaggi AMQP dal DB PostgreSQL(PUBLISH);
- l'ascolto da parte di RabbitMQ di notifiche da parte del DB PostgreSQL(NOTIFY).

In particolare per ricevere queste notifiche, è stato necessario poter installare i file idonei allo scopo.
Un tipo di scambio(exchange) di RabbitMQ traduce i messaggi PostgreSQL NOTIFY in messaggi AMQP e li pubblica in code associate. Il canale di messaggi PostgreSQL NOTIFY viene utilizzato come chiave di routing per il messaggio utilizzando lo scambio diretto.
Una volta che i file sono stati installati in maniera corretta, attraverso il pannello di configurazione di RabbitMQ, consultabile all'indirizzo http://192.168.99.100:8080, è stato necessario procedere alla dichiarazione di un exchange di tipo *x-pgsql-listen* di nome "*notifications*", come indicato nella figura sottostante:

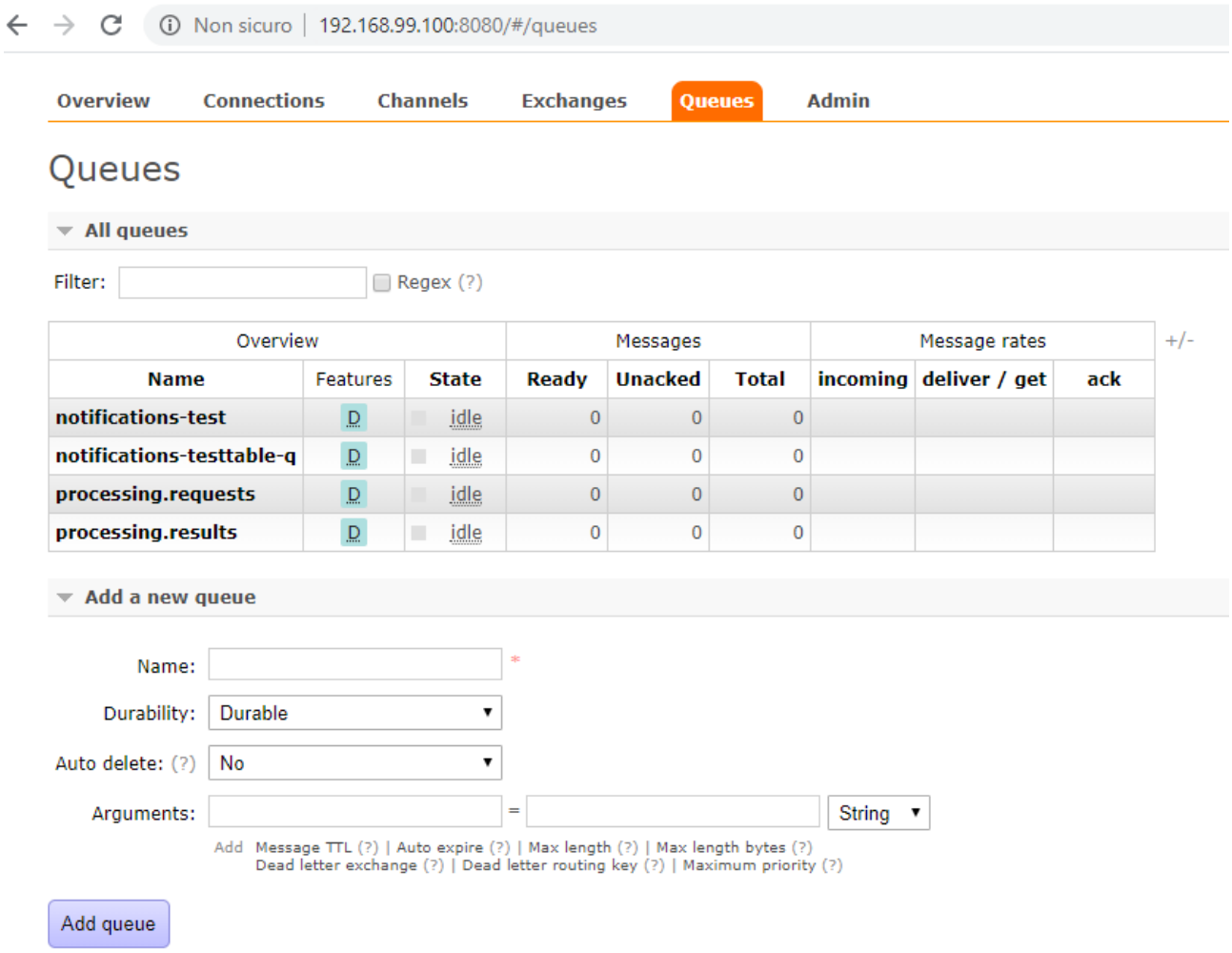


Vi sono altri modi per la dichiarazione di un exchange, come la sua configurazione all'interno del file *rabbitmq.config*, opportunamente creato dove si trova il file eseguibile di RabbitMQ o tramite dichiarazione di una policy. La policy rappresenta un modo molto flessibile per permettere lo scambio e può essere utile quando non si ha molta esperienza o si è alle prime armi con il plugin.

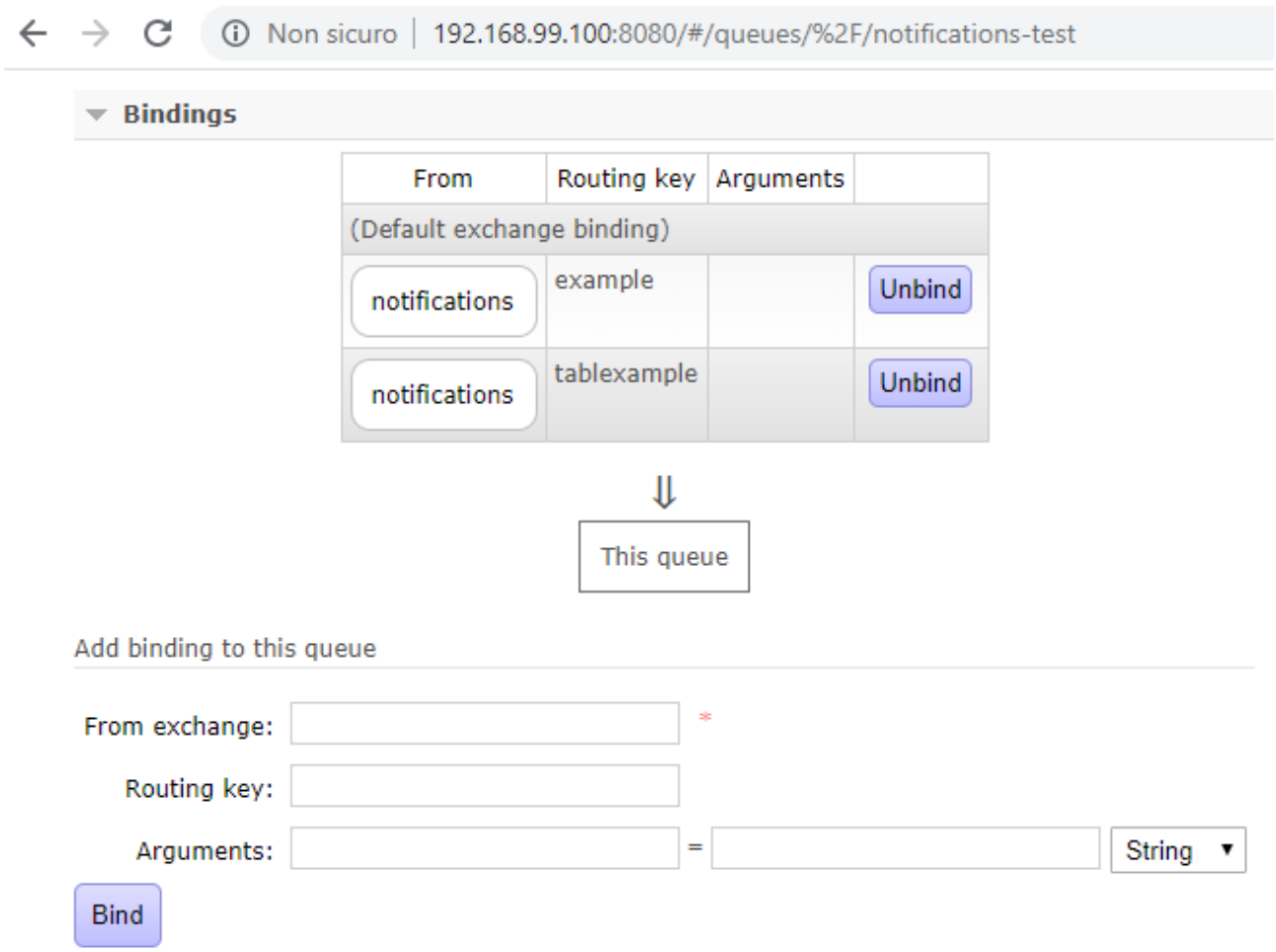
Una policy fondamentale è stata creata per quanto riguarda la connessione a PostgreSQL, specificando chiaramente un'espressione regolare da abbinare all'exchange, oltre ai parametri come username, password, nome del DB sul quale si lavora, host e porta dedicata:



Una volta creata questa policy, non si sa ancora se questa possa permettere una connessione valida per un possibile scambio di messaggi.
Come visto sopra, si sono viste le fasi per la creazione di un exchange. Questo può connettersi a PostgreSQL ma non è ancora in grado di ascoltare delle notifiche. Per poter iniziare ad ascoltare, è necessario associare all'exchange una chiave di routing(routing key), la quale corrisponde alla stringa del canale di notifica di PostgreSQL.
L'ultimo passaggio per permettere l'ascolto(o LISTEN exchange), è la creazione di una coda di messaggi(usata prevalentemente come test) nella quale verranno inviate le notifiche. Una volta creata questa coda, è possibile creare un'associazione(binding) tra l'exchange e la routing key. Dopodiché sarà possibile per l'exchange connettersi a PostgreSQL, eseguendo una LISTEN, quindi è possibile “catturare” tutte le notifiche che saranno inviate sul canale in uso.



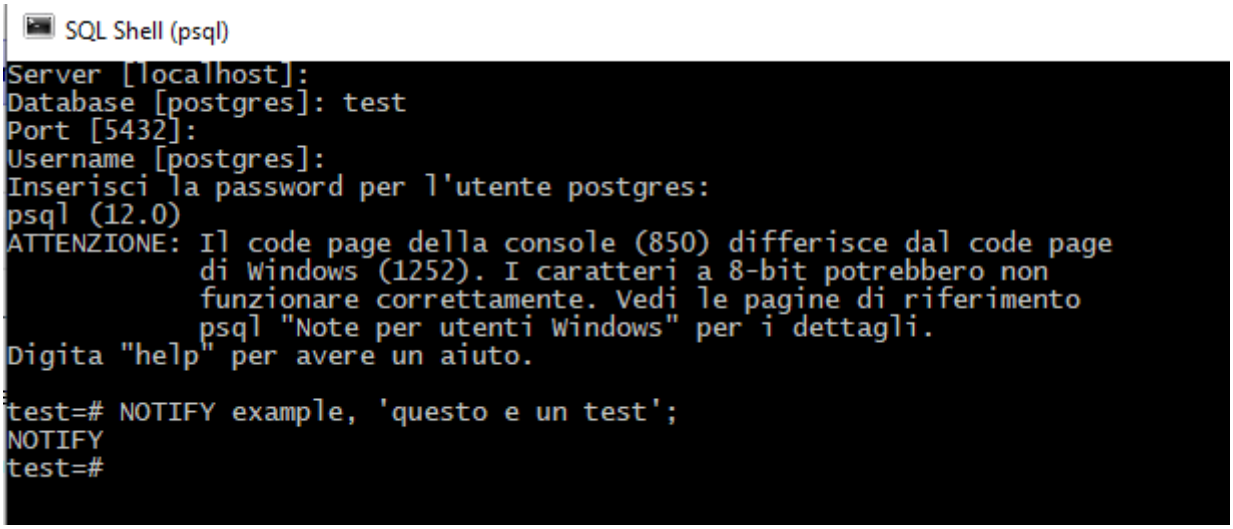
creazione code, nello specifico sono state create varie code per vari test di prova(notifications-test e notifications-testtable-q), ma quelle utilizzate per il progetto sono processing.requests e processing.results



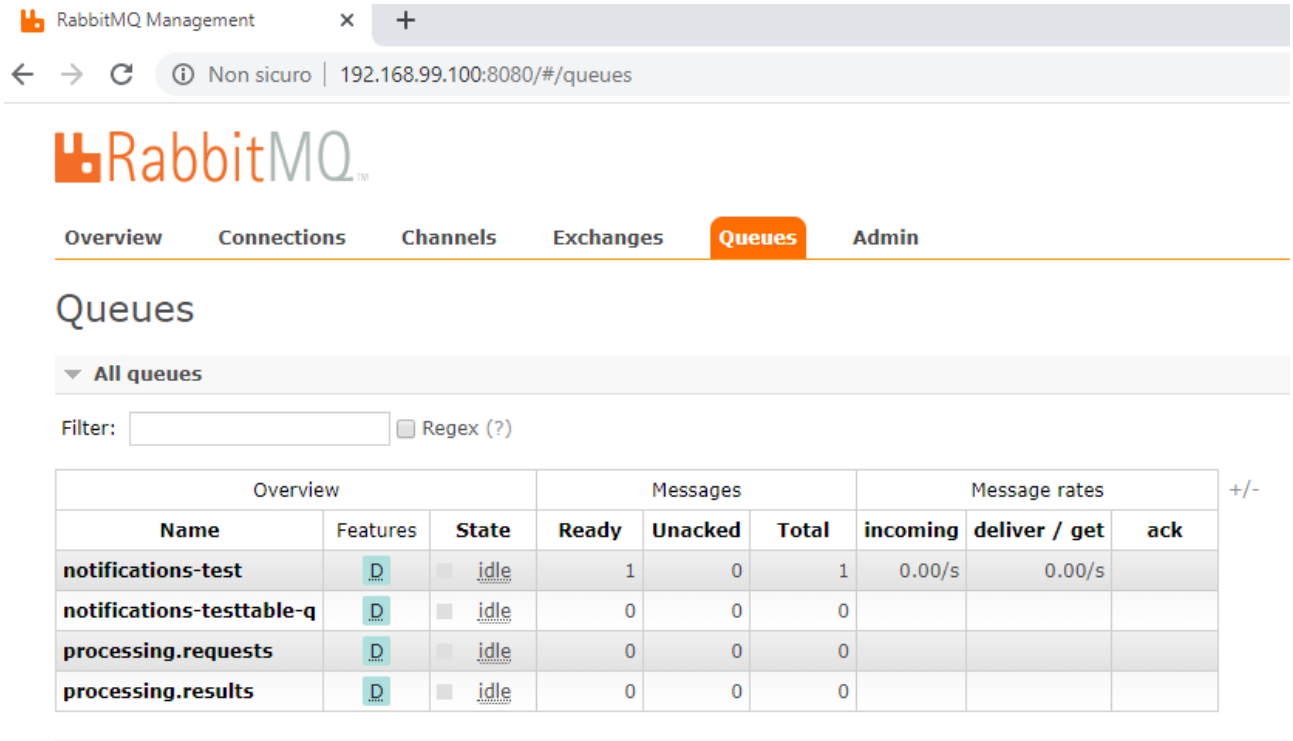
creazione delle associazioni(binding), nello specifico quelle dichiarate per la coda notifications-test.

Si nota come è possibile poter aggiungere ulteriori associazioni o chiavi di routing per la stessa coda

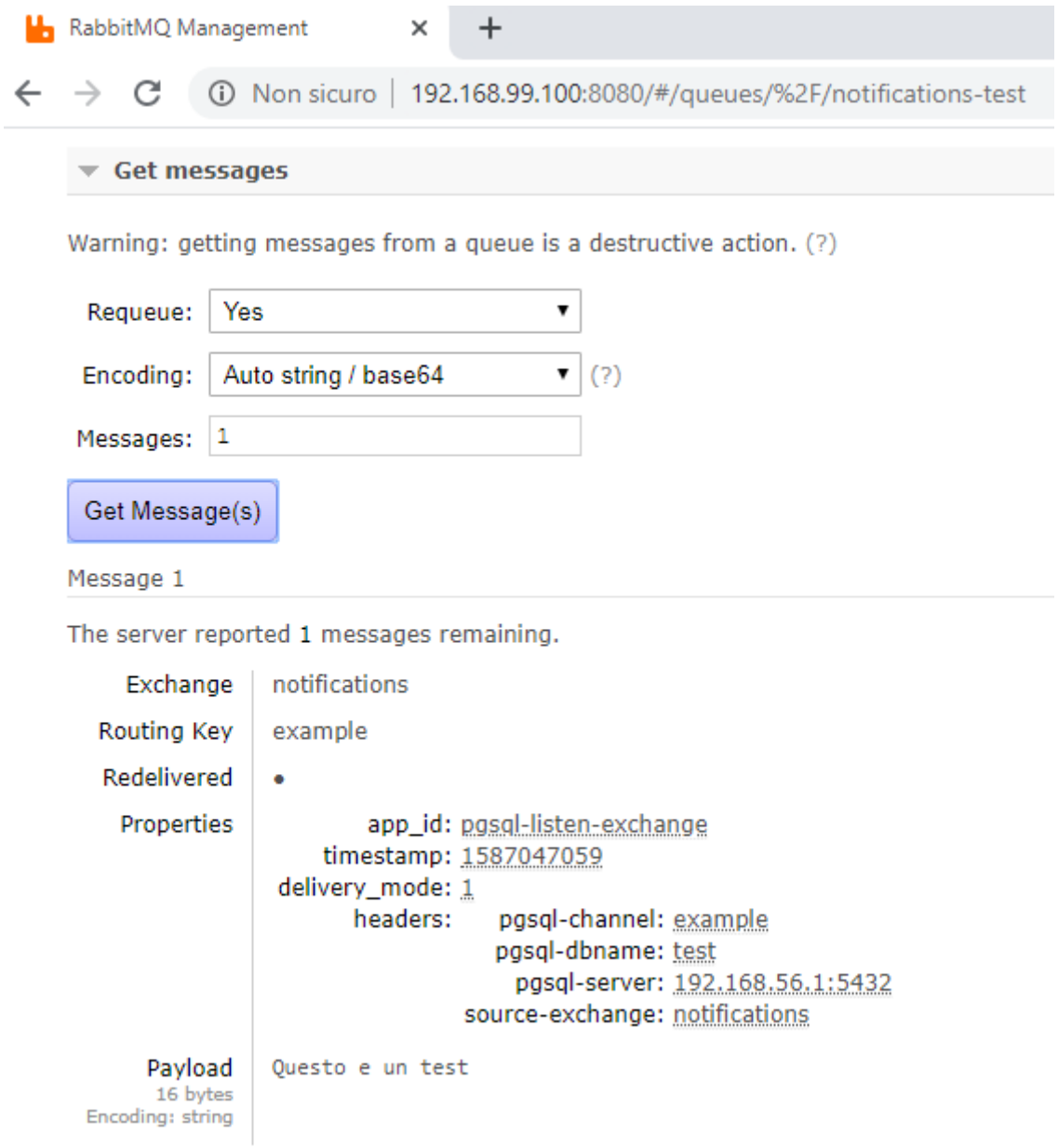
Notiamo che in questo modo l'exchange è pronto per la connessione a PostgreSQL, eseguendo una LISTEN e può ricevere le notifiche inviate sul canale. Possiamo verificare il funzionamento di un exchange attraverso uno statement SQL NOTIFY. Per questa prova possiamo usare psql da un terminale, collegandoci al DB di lavoro, come rappresentato in figura:



Si noterà, che all'inserimento della NOTIFY dove si è dovuto specificare l'exchange(di nome example), arriverà il messaggio(contenuto tra apici), nella coda “notifications-test”, evidenziato nella tabella del pannello di controllo di RabbitMQ, come notato nella seguente immagine:

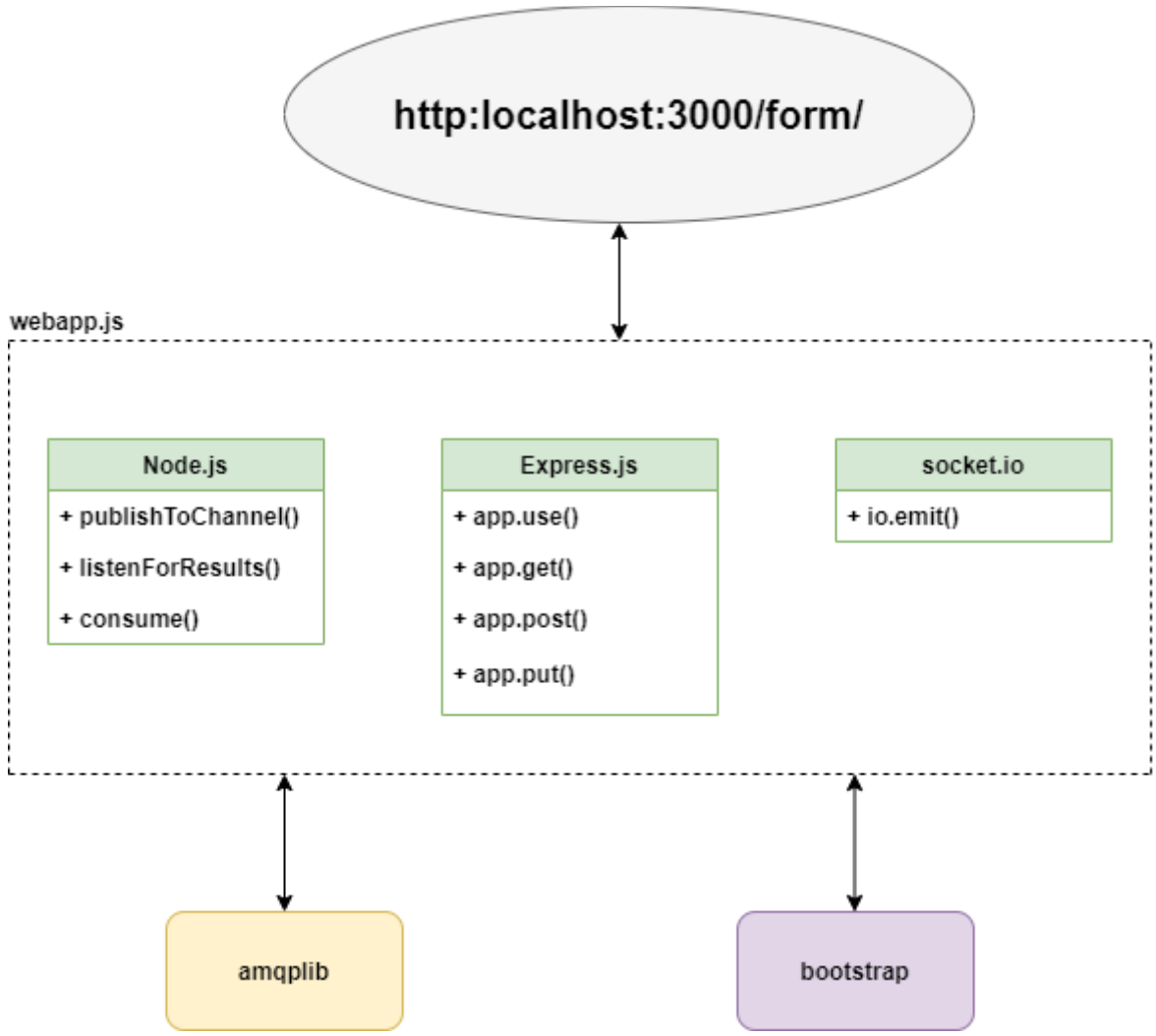


Dettagliatamente, all'interno della coda stessa, si potrà vedere cosa contiene il messaggio, attraverso l'attributo Payload, ed è proprio il messaggio precedentemente immesso tra gli apici:



Si noti che per il salvataggio delle informazioni di un utente sul database si è dovuto procedere alla creazione di trigger[13] in SQL. Sono procedure particolari che si verificano al termine di un determinato evento. Nel caso del progetto assegnato si sono creati per l'exchange "*result*" dopo ogni operazione di inserimento, aggiornamento o cancellazione del DB. Vedremo più avanti che le operazioni effettuate saranno tramutate in messaggi in coda.

Diagramma delle classi



Il codice dell'applicazione *webapp.js* sostanzialmente come si vede dallo schema precedente, si divide in più parti. Sono state utilizzate diverse librerie e/o framework.

Descrizione delle classi(o funzioni) divise per componenti

Node.js

publishToChannel(): è una funzione di utilità che pubblica i messaggi su un canale dedicato. Prende in input il canale(*channel*), creato in precedenza da RabbitMQ; la *routingKey*, per scegliere la coda; l'*exchangeName*, il tipo di exchange configurato in RabbitMQ e *data*, ovvero i dati del messaggio. La funzione che pubblicherà il messaggio sulla coda di RabbitMQ sarà *channel.publish.*

listenForResults(): è una funzione che si mette in ascolto sulla coda dei messaggi in modalità asincrona(*async*). Insieme ad *await* sono coppia di valori chiave di Node.js i quali consentono la scrittura di codice in modo procedurale senza dover fare affidamento su altro codice scritto. Si connette alla coda RabbitMQ ed al canale analizzando un messaggio alla volta, richiamando la funzione *consume()*.

consume(): è la funzione che, appunto, "consuma" i messaggi dalla coda RabbitMQ. Utilizzando l'*exchange* e la *routingKey* viene letto il dato contenuto nel messaggio stesso, inviandolo in tempo reale tramite *io.emit()* (funzione di socket.io) alla pagina *stato.html*.

Express.js

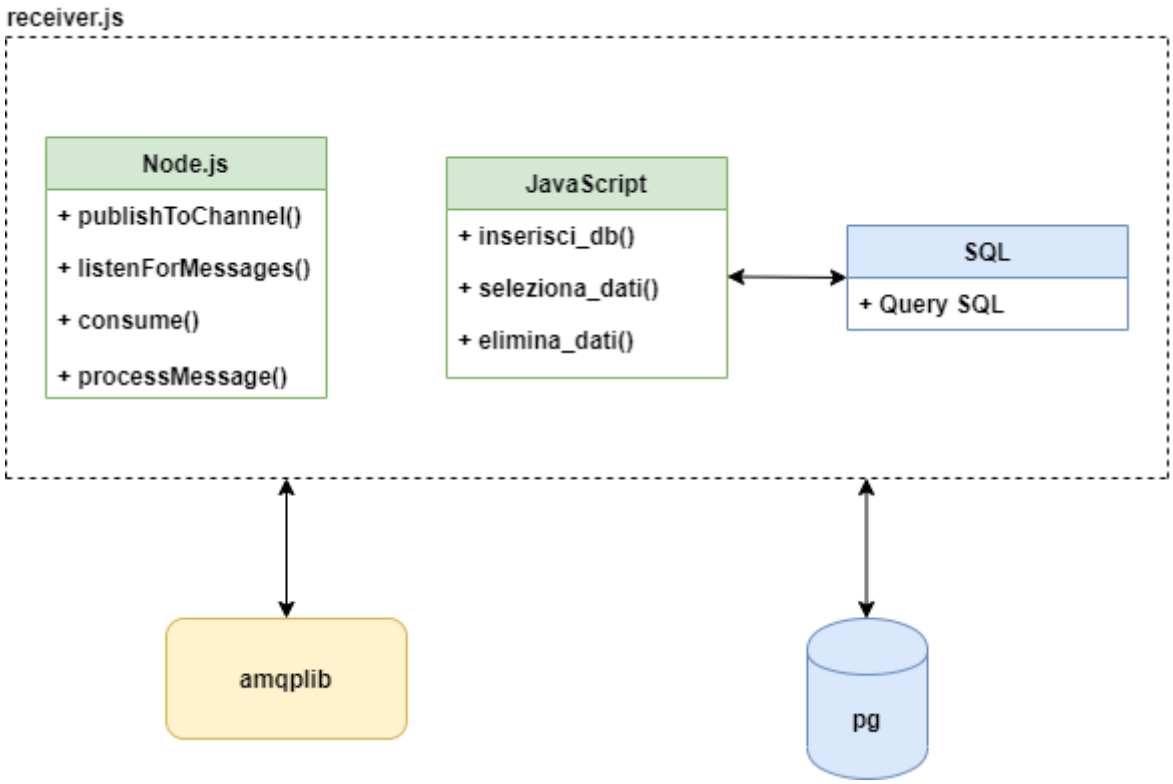
app.use(): è stata usata per includere nella pagina il framework bootstrap.

app.get(): tramite get si sono sviluppati gli endpoint che tramite il routing serviranno le pagine di form e di stato(*iscrizione.html* e *stato.html*).

app.post(): notiamo che il route *'/api/v1/processData'* è la risposta, o meglio il percorso, che verrà generato dal form. In esso saranno contenuti i dati della compilazione di un utente, generando una richiesta di inserimento(*INSERT*) dei suddetti dati. Questi ultimi verranno pubblicati su RabbitMQ ed in tempo reale nella pagina di stato.
Analogamente il route *'/cancella/:id'* contiene richieste di eliminazione(*DELETE*) dei dati.

socket.io

io.emit(): è una funzione che invierà dei dati in tempo reale. In questo caso la pagina *stato.html* è in ascolto da quello che verrà svolto da webapp.js, tra cui l'inserimento o l'eliminazione di dati.



Di seguito illustriamo il codice dell'applicazione receiver.js. Ha usato principalmente la libreria *pg* per la connessione al DB PostgreSQL.

<u>Descrizione delle classi(o funzioni) divise per componenti</u>

Node.js

publishToChannel(): abbiamo già visto l'utilità di questa funzione. Pubblica il messaggio su un canale dedicato, prendendo in input la coda, l'exchange, la routing key ed i dati del messaggio.

listenForMessages(): è una funzione in modalità asincrona, si connette a RabbitMQ analizzando un messaggio alla volta dal canale di ascolto. Crea inoltre un secondo canale in modo da confermare i dati ricevuti. Richiama poi la consume() per, appunto, “consumare” i messaggi.

consume(): è la funzione che, appunto, “consuma” i messaggi dalla coda RabbitMQ. In questo caso, “consumerà” dalla coda *processing.requests*. Questa è la coda di messaggi “collegata” alle operazioni che vengono eseguite direttamente sul DB(opportunamente scelta come “coda di richieste”). In base ai possibili casi verranno annotate le operazioni di selezione, inserimento o eliminazione dal DB stesso.

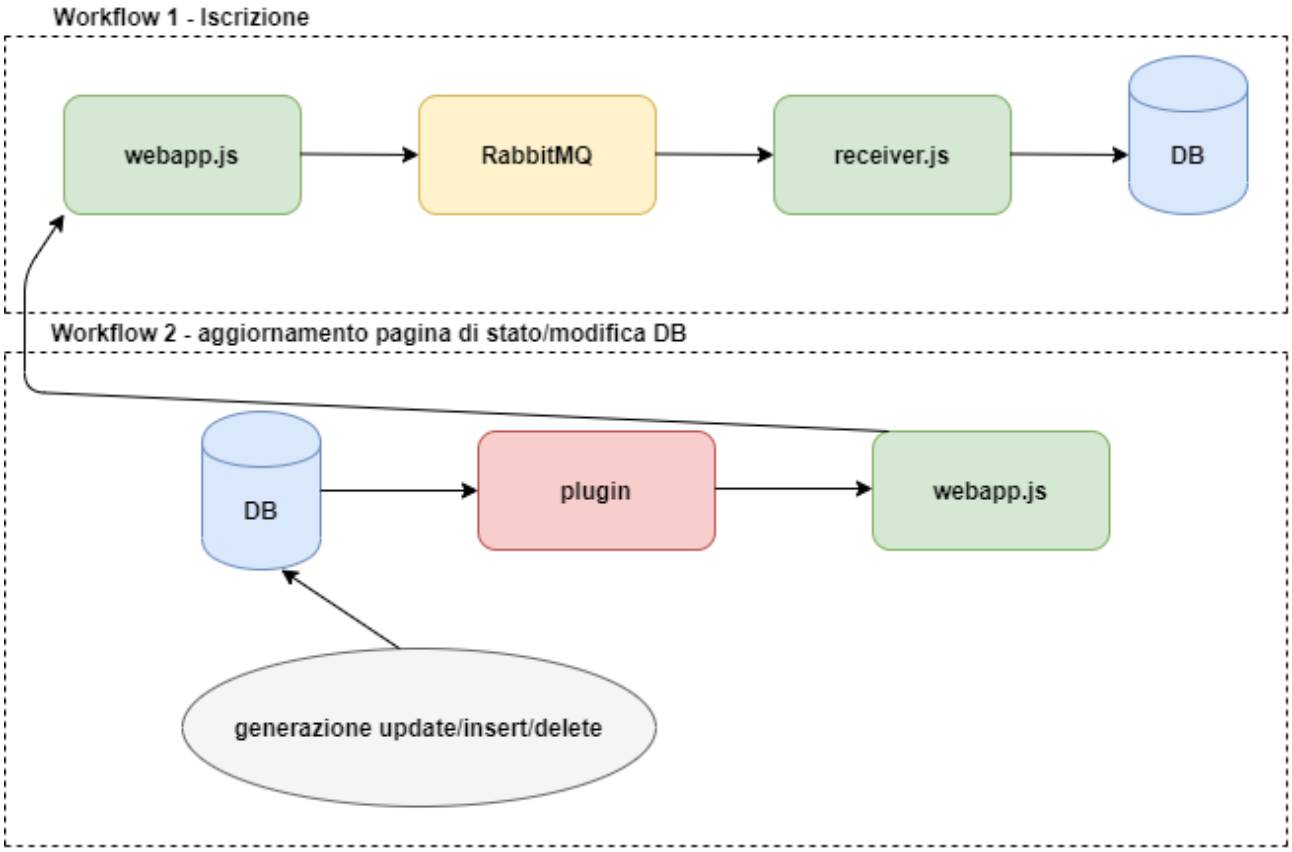
processForMessage(): aggiorna lo stato da “*pending*” a “*completed*”, secondo l'operazione svolta sul DB in tempo reale, visibile nella pagina di *stato.html*.

JavaScript/SQL

inserisci_db(), seleziona_dati(), elimina_dati(): non sono altro che le funzioni apposite per l'integrazione con il DB. L'inserimento viene svolto attraverso la compilazione dei dati di un utente via form, con il particolare codice SQL 'INSERT INTO'. La selezione dati è svolta tramite codice SQL 'SELECT'. L'eliminazione è svolta tramite codice SQL 'DELETE', verranno eliminati tutti i dati relativi ad un particolare id della tabella utente.

Descrizione interfaccia utente

Analizziamo con esempi pratici come funziona il sistema proposto, basandosi anche su workflow che mostrano l'iscrizione di un utente ed operazione di aggiornamento della pagina *stato.html* ove si modifichi il DB:



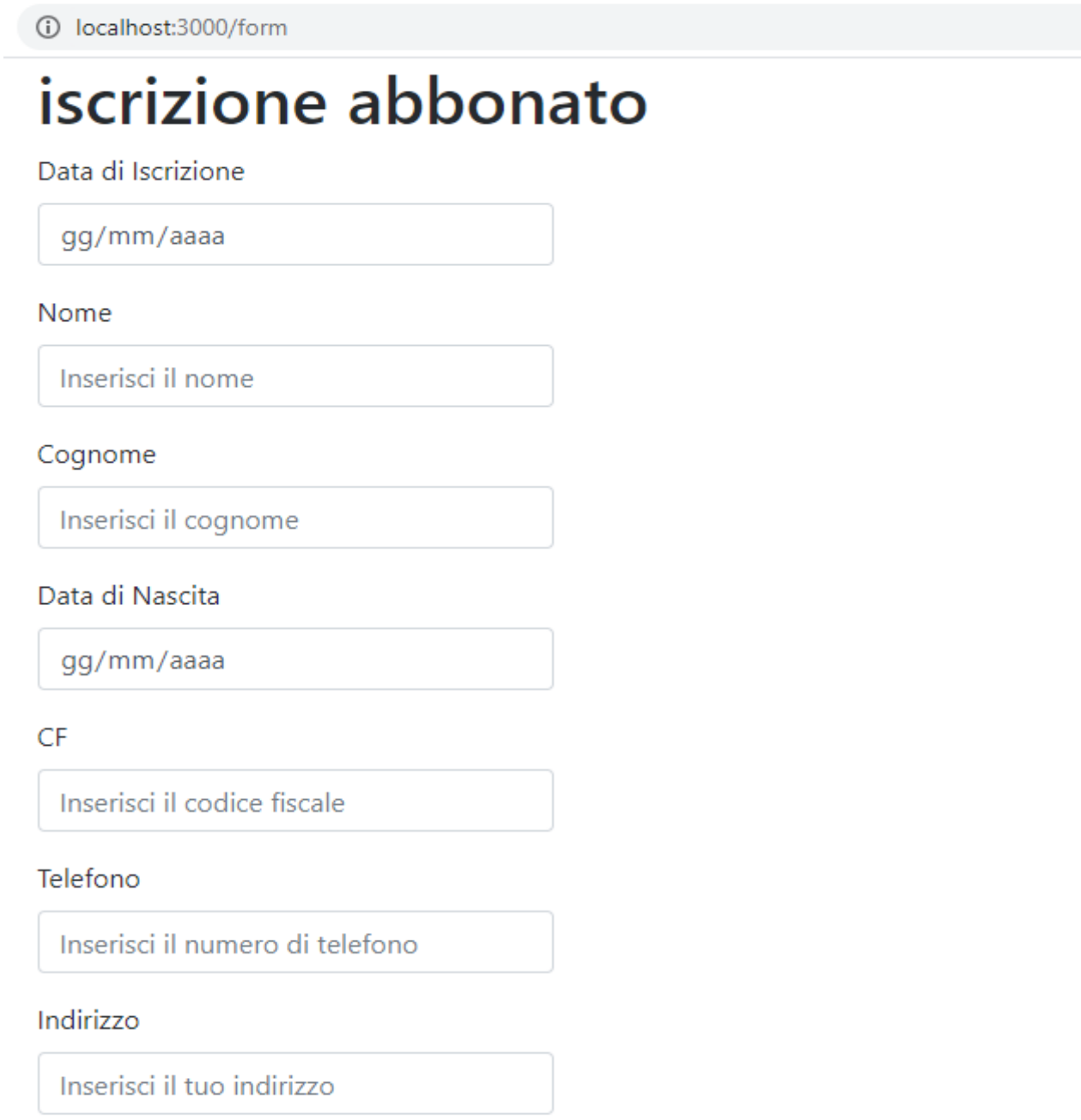
Per prima cosa, tramite Docker, si avvia la VM:

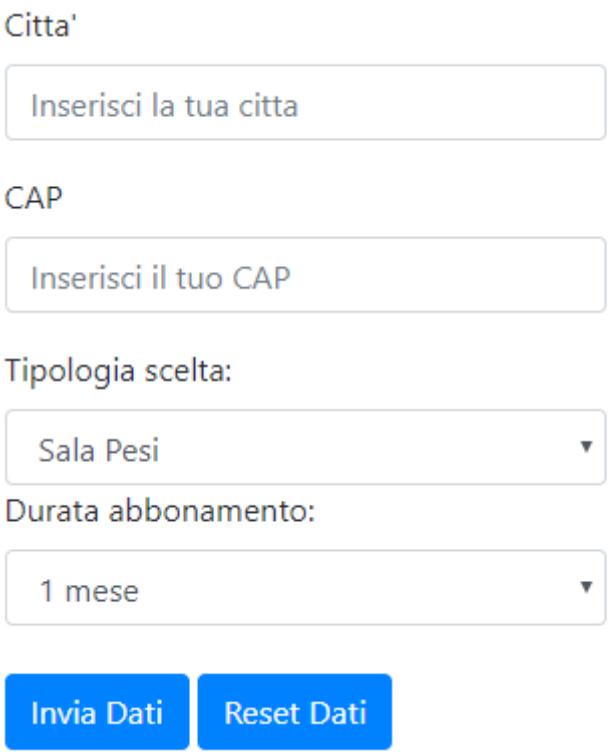
```
boss@DESKTOP-M6USILV MINGW64 /c/Program Files/Docker Toolbox
$ docker start manage
manage
```

Da un terminale, si avvia il server locale di Express.js, dopodiché da browser si apre la pagina di form all'indirizzo *http://localhost:3000/form*:

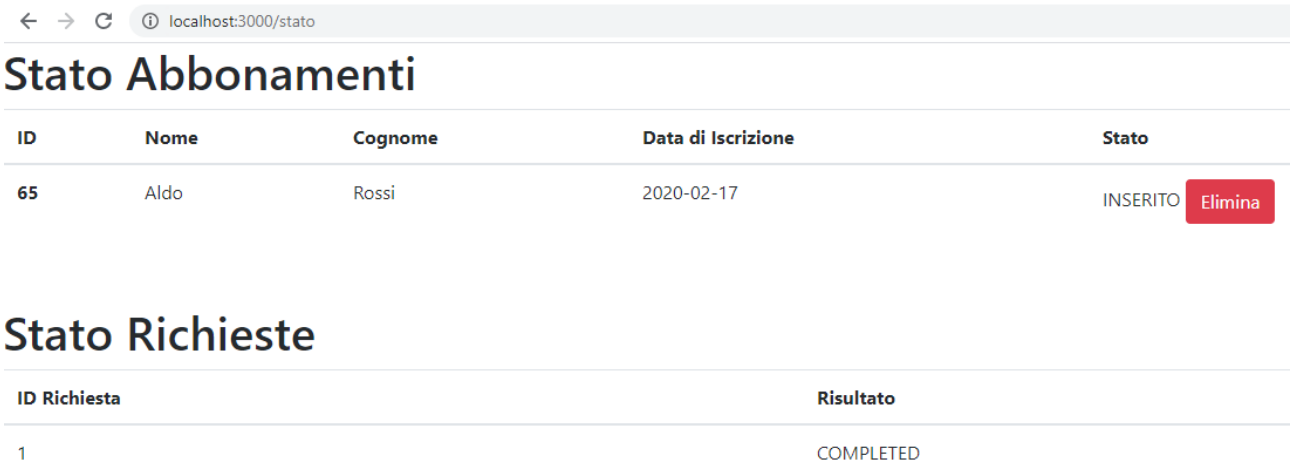
```
PS C:\Users\boss\Desktop\LaboratorioRetiMateriale\codice> node
Listening on port 3000.
```

WorkFlow 1 – Iscrizione (INSERT)

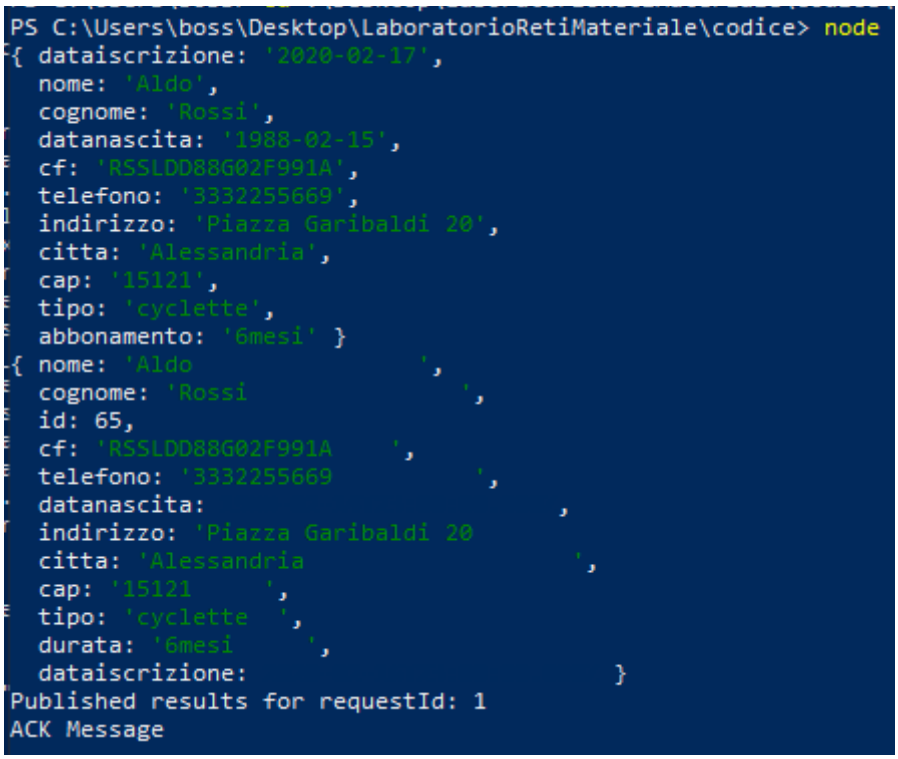


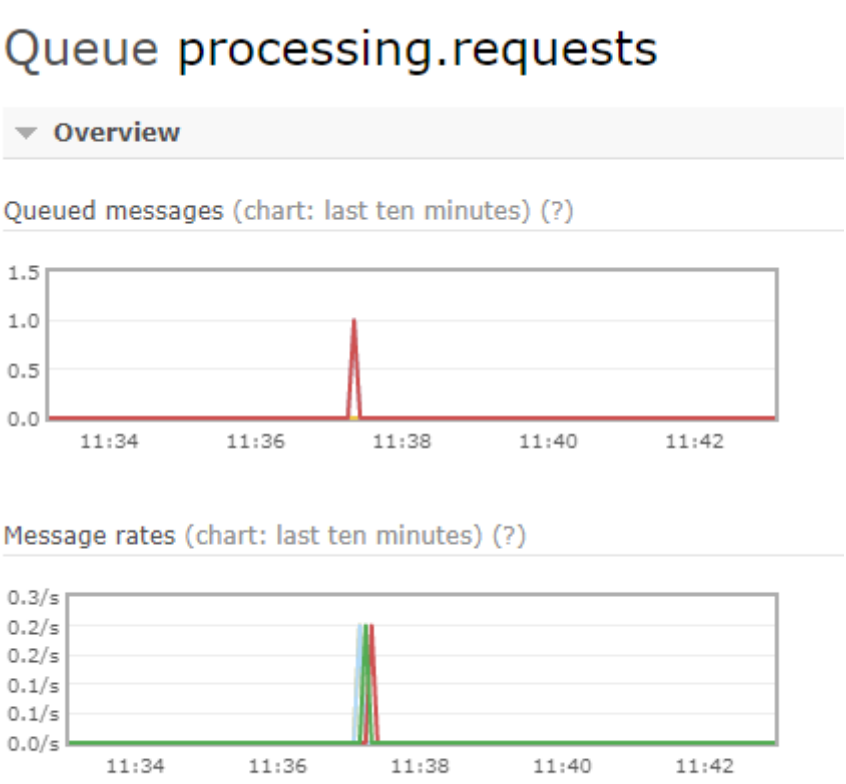


il form permette l'immissione di dati che verranno salvati successivamente sulla tabella del DB (esempio svolto con “Aldo Rossi”).



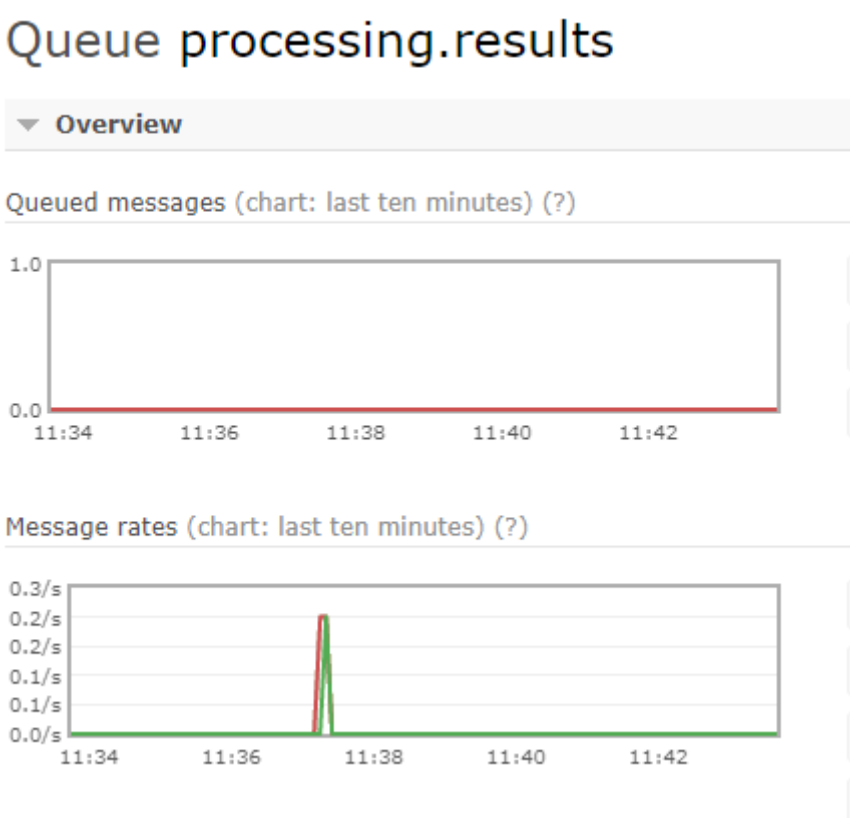
Vediamo che la pagina di stato ha ricevuto in tempo reale l'inserimento dei dati. Andiamo a vedere cosa succede in RabbitMQ:





Si è riusciti ad ottenere il messaggio contenente i dati inseriti dal form dalla coda *processing.requests* attraverso l'esecuzione dell'applicazione *receiver.js*

```
PS C:\Users\boss\Desktop\LaboratorioRetiMateriale\codice> node
Listening on port 3000.
Published a request message, requestId: 1
prima dell IF {"table" : "utente", "id" : 65, "nome" : "Aldo",
siamo dentro IF { table: 'utente',
  id: 65,
  nome: 'Aldo',
  cognome: 'Rossi',
  dataiscrizione: '2020-02-17',
  type: 'INSERT' }
emetto segnale di operazione avvenuta ed invio i dati
```



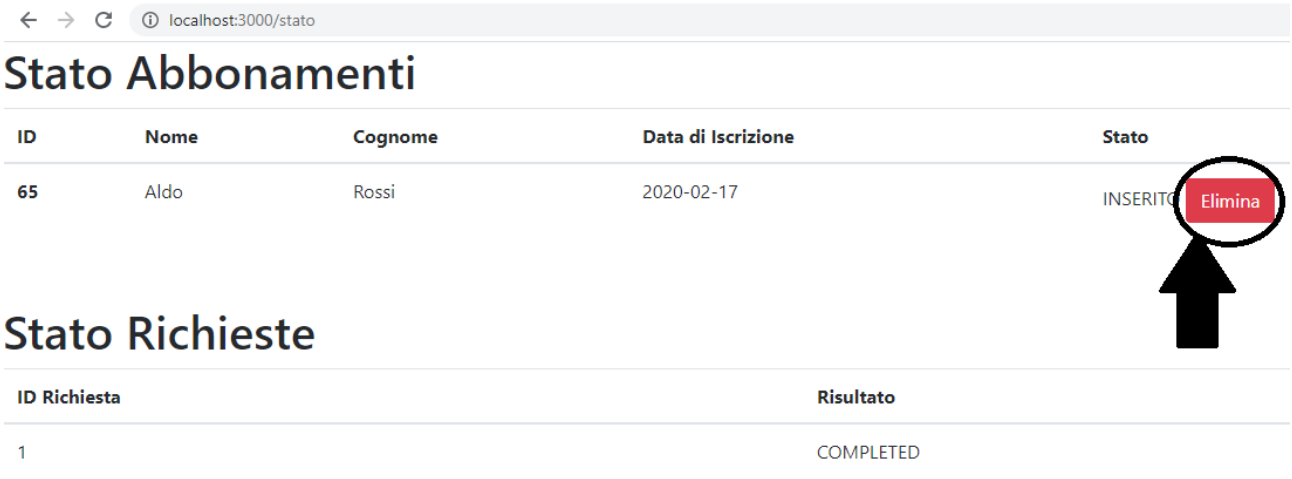
Si è riusciti ad ottenere il messaggio contenente i dati inseriti dal form dalla coda *processing.results* attraverso il codice dell'applicazione *webapp.js*

```
test=# SELECT * FROM utente WHERE id='65';
        nome         |       cognome        | id |         cf         |      telefono       | datanascita |
        citta        |    cap     |    tipo    |   durata   | dataiscrizione
---------------------+--------------------+----+--------------------+---------------------+-------------+-
-------------------------+------------+------------+------------+----------------
 Aldo                | Rossi              | 65 | RSSLDD88G02F991A   | 3332255669          | 1988-02-15  |
ssandria             | 15121      | cyclette   | 6mesi      | 2020-02-17
(1 riga)
```

notiamo come anche da terminale PSQL che l'inserimento in tabella è andato a buon fine sul database *test,* sulla tabella *utente* in uso per lo scopo finale.

Workflow 2 – Aggiornamento pagina di stato/modifica DB(DELETE)

Se si vogliono eliminare dalla tabella tutti i dati relativi ad “Aldo Rossi”, quindi effettuare una DELETE, notiamo che nella pagina *stato.html* vi è un bottone rosso “*elimina*”:



Se vi clicchiamo sopra, riusciamo ad eliminare dalla tabella tutti i dati relativi all'utente “Aldo Rossi”, come confermato sia dai messaggi sulla coda RabbitMQ *processing.results*, sia dalla tabella del DB, il tutto visibile attraverso l'esecuzione di receiver.js:

```
id da eliminare { op: 'DELETE', requestId: 8, userid: '65' }
data { op: 'DELETE', requestId: 8, userid: '65' }
[]
Published results for requestId: 8
```

richiesta di DELETE con id=8, completata come visibile in pagina *stato.html*:

Notiamo che nel DB non vi è più traccia dei dati di “Aldo Rossi”:

```
test=# SELECT * FROM utente WHERE id='65';
 nome | cognome | id | cf | telefono | datanascita | indirizzo | citta | cap | tipo | durata | dataiscrizione
------+---------+----+----+----------+-------------+-----------+-------+-----+------+--------+----------------
(0 righe)

test=#
```

Problemi riscontrati e soluzioni

Come scritto in precedenza, l'uso del plugin utilizzato è limitato, in quanto compatibile fino alla versione 3.5 di RabbitMQ. E' stato opportuno definire quindi, un'immagine Docker(da dockerfile creato) per la configurazione e l'esecuzione di un broker RabbitMQ 3.5.

Conclusione

Abbiamo visto con questa sperimentazione a che cosa serve un broker, unitamente ad un plugin per l'interazione con un DB, rendendolo così interfacciabile con più applicazioni suscitando interesse, ottenendo informazioni un po' più dettagliate sotto forma di scambio di messaggi.

Bibliografia

[1]: https://github.com/gmr/pgsql-listen-exchange
[2]: https://getbootstrap.com/
[3]: https://nodejs.org/it/about/
[4]: https://expressjs.com/it/
[5]: https://www.docker.com/
[6]: https://www.rabbitmq.com/
[7]: https://www.pgadmin.org/
[8]: https://socket.io/
[9]: https://node-postgres.com/
[10]: https://www.npmjs.com/package/amqplib
[11]: https://www.cloudamqp.com/blog/2015-05-19-part2-2-rabbitmq-for-beginners_example-and-sample-code-node-js.html
[12]: https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/advanced-message-queuing-protocol-amqp/
[13]: https://www.html.it/pag/31837/i-trigger1/