Documentazione

Indice

Requisiti …………………………………………………………………………………… pag. 1

Tecnologie utilizzate ……………………………………………………………………… pag. 1

Design Pattern ………………………………………………………………………………. pag. 1

Message Broker ………………………………………………………………………….... pag. 2

Servizi ……………………………………………………………………………………………. pag. 2

Diagramma ER ………………………………………………………………………………. pag. 3

Dipendenze tra i servizi …………………………………………………………………. pag. 4

Scambio di messaggi Redis ……………………………………………………………. pag. 4

Logica del progetto ………………………………………………………………………. pag. 4

Requisiti

R1. Richiesta di invio e Gestione email

R2. Richiesta di invio e Gestione SMS

R3. Richiesta di invio e Gestione Push Notification

Tecnologie utilizzate

Spring Framework

Spring Framework fornisce un modello di programmazione e configurazione completo per le moderne applicazioni enterprise Java-based su qualsiasi tipo di piattaforma di deployment.

Un elemento chiave di Spring è il supporto infrastrutturale al livello applicazione: Spring si focalizza sulle “fondamenta” di un’applicazione enterprise così che gli sviluppatori possono focalizzarsi su business logic a livello applicazione, senza necessitare di specifici deployment environments

Spring Data

Lo scopo di Spring Data è quello di fornire un modello di programmazione per l’accesso dei dati basato su Spring familiare e coerente pur mantenendo i tratti speciali del data store sottostante. Semplifica l’utilizzo delle tecnologie per l’accesso dei dati, database relazionali e non relazionali, map-reduce frameworks, e data service basati su cloud.

Spring Boot

Spring Boot viene utilizzato per semplificare la creazione di applicazioni basate su Spring in modo tale da renderlo più rapido

Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) è una soluzione di messaggistica cross-platform che permette di mandare messaggi senza costo. Perché firebase?

Twilio

È un’API di Java che ha tra le sue feature quella dell’invio di notifiche SMS.

Redis

Redis è una struttura dati in-memory open source, usato come database, cache, e message broker. Redis fornisce diverse strutture dati: strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs, geospatial indexes, e streams.

**Tutte le operazioni sono atomiche**, quindi in caso di accessi concorrenti da parte di più client, i dati forniti risulteranno sempre aggiornati;

Spring mail

Libreria utilizzata per l’implementazione dell’effettivo invio della mail.

Gmail

In particolare, viene configurato ed utilizzato un account gmail per l’invio delle notifiche mail.

Nota: per la configurazione smtp l’account deve avere abilitata la verifica in due passaggi altrimenti non è possibile accedere alla sezione che serve per creare la key di autenticazione per applicazioni terze.

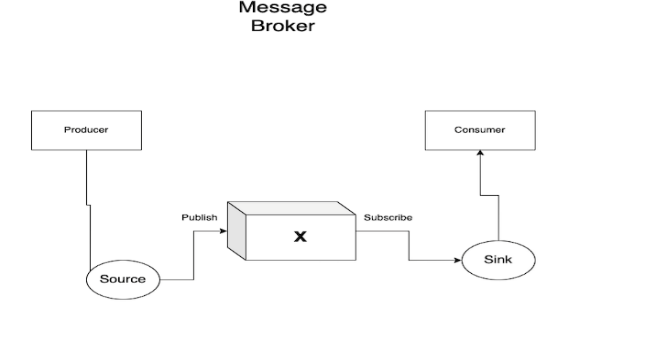
Design Pattern

Il Design Pattern utilizzato è IoC in quanto è quello del framework Spring.

A differenza dei design pattern tradizionali dove gli oggetti definiscono le loro dipendenze solamente tramite gli argomenti del costruttore, argomenti del factory method e il container fa inject di queste dipendenze quando viene creato il bean; nell’ IoC il processo è l’inverso: è il bean a controllare l’istanziazione o allocazione delle sue dipendenze.

Spring implementa l’IoC tramite la **Dependency injection**. La **DI** prevede che tutti gli oggetti all’interno della nostra applicazione accettino le dipendenze, ovvero gli (altri) oggetti di cui hanno bisogno, tramite costruttore o metodi setter. Non sono quindi gli stessi oggetti a creare le proprie dipendenze, ma esse vengono iniettate dall’esterno.

Message Broker

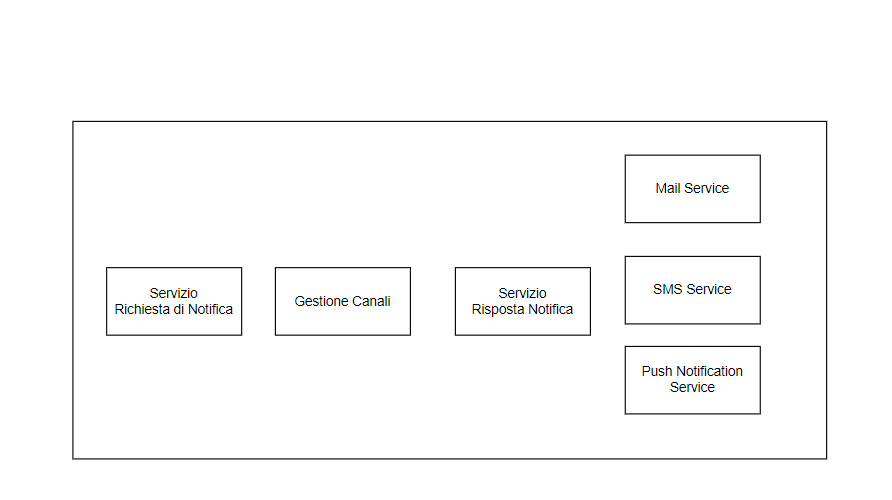


Il message broker è un modello architetturale per lo scambio di messaggi, che fa da intermediario tra il mittente (Producer) ed il destinatario (Consumer).

**Publish–subscribe** è un tipo di scambio dove il mittente del messaggio(publisher) non programma che un messaggio sia inviato direttamente ad un ricevitore specifico(subscriber), ma categorizza i messaggi pubblicati in delle classi, senza sapere chi siano i subscriber.

In questo caso verrà utilizzato Redis come message broker.

Servizi



**Service Richiesta di Notifica**: Micro-servizio che si occupa di gestire l’acquisizione e la gestione delle richieste di notifica

**Gestione Canali**: Micro-servizio che si occupa della gestione dei canali di comunicazione e della modifica dello stato di richiesta e risposta di notifica

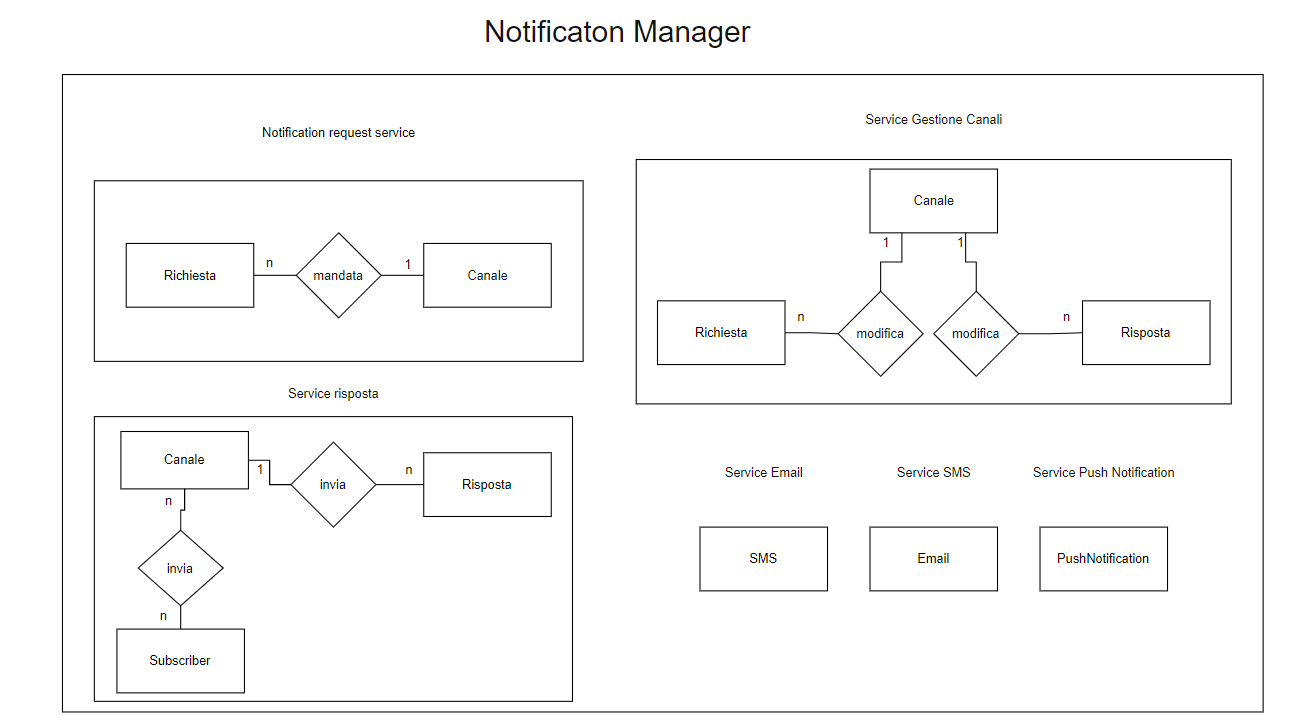
**Service Risposta Notifica**: Micro-servizio che si occupa della creazione della Risposta alla richiesta di notifica

**Email Service**: Micro-servizio che si occupa nel dettaglio della creazione della email contenente tutte le informazioni della notifica a seguito di una corretta formazione della risposta di notifica

**SMS Service**: Micro-servizio che si occupa nel dettaglio della creazione dell’SMS contenente tutte le informazioni della notifica a seguito di una corretta formazione della risposta di notifica

**Push Notification Service**: Micro-servizio che si occupa nel dettaglio della creazione delle Push Notification contenente tutte le informazioni della notifica a seguito di una corretta formazione della risposta di notifica

Diagramma ER



**Richiesta**: id, stato, messaggio, tipo, data, ora, priorita, multicast, id\_canale

**Canale**: nome, tipo, info, num\_sub

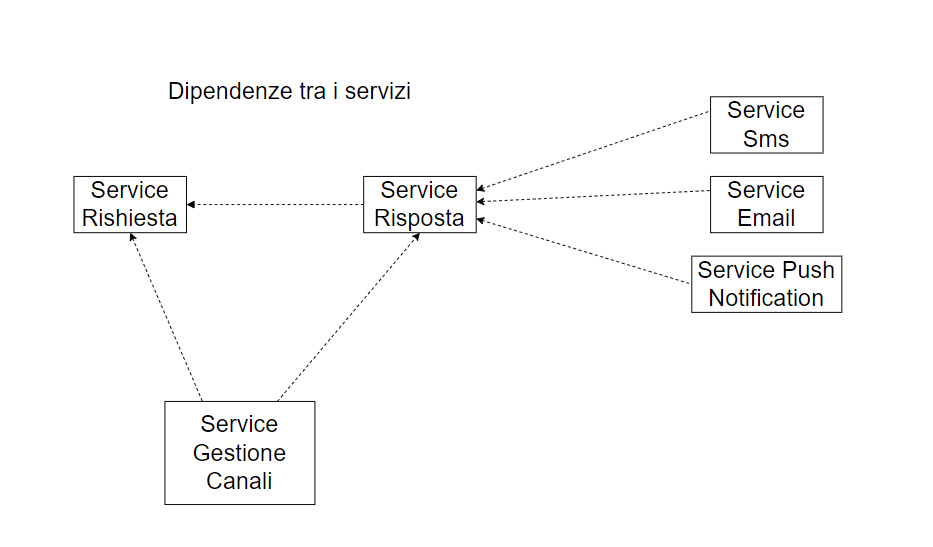
**Risposta**: id, azione, messaggio, stato, id\_canale, id\_richiesta

**Utente**: id, email, num\_tel, token, id\_canale

**Email**: id, oggetto, testo, path\_allegato, allegato, id\_risposta

**Sms**: id, messaggio, id\_risposta

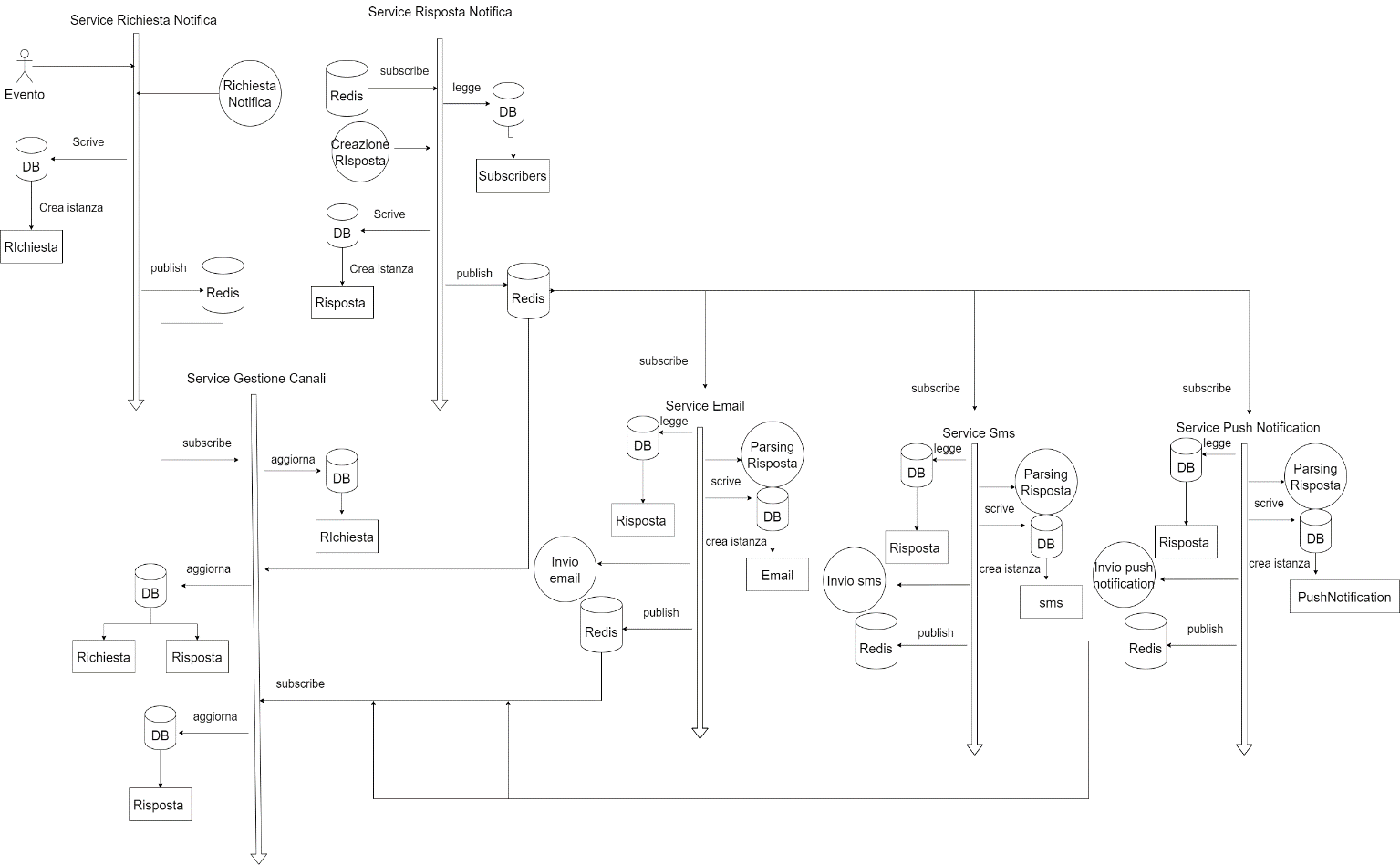
**PushNotification**: id, topic, title, body, id\_risposta



Scambi di messaggi Redis

La struttura dei messaggi redis che verranno scambiati dall’applicazione è formata da una coppia chiave-valore dove come chiave hanno *il tipo* (richiesta o risposta) e come valore hanno *l’identificativo* della richiesta o risposta; tali messaggi verranno poi mandati su un canale di comunicazione in modo tale che tutti i subscriber potranno riceverli. La scelta di utilizzare solamente l’identificativo è data dal fatto che tutte le altre informazioni necessarie per creare e gestire la notifica sono già presenti all’interno del database.

Logica del progetto



**Service Richiesta Notifica**: A seguito di un evento scatenante viene creata una *Richiesta di Notifica*, la cui istanza viene successivamente salvata nel *Database* nella tabella *Richiesta*. Una volta terminata questa operazione viene fatta una publish su un canale redis *,* che servirà sia al **Service Gestione Canali** che al **Service Risposta Notifica.**

**Service Gestione Canali:** Rimane sempre in ascolto sui canali redisper aggiornare lo stato delle tabelle di *Richiesta* e *Risposta.*

**Service Risposta Notifica:** Rimane in ascolto su un *canale redis* e aspetta che venga fatta una *Richiesta di Notifica*; appena ne individua una cerca nella tabella del *Database Subscribers* tutte le istanze legate a quello specifico canale, successivamente viene creata un’istanza di *Risposta Notifica* che viene salvata nella tabella del Database *Risposta*; in fine viene fatta una *push* sul *canale redis* che servirà a **Service Email**, **Service Sms** e **Service Push Notification** per creare la specifica notifica che verrà inviata.

**Service Email**, **Service Sms** e **Service Push Notification:** Rimangono in ascolto su un canale redis, specifico per il loro tipo di notifica, aspettando un *Risposta Notifica* di cui poi prenderanno un’istanza nella tabella *Risposta* per poter generare un’istanza nella loro tabella specifica (*Email, Sms, PushNotification*). Successivamente procedono con l’invio della notifica appena creata e in fine fanno una *publish* su un *canale redis*.