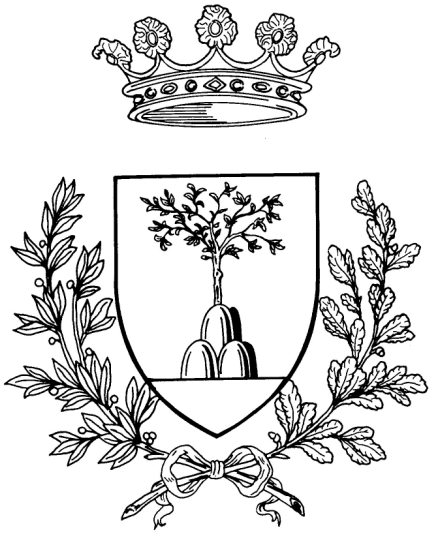
UNIVERSITA' DEGLI STUDI DI FERRARA

FACOLTA' DI INGEGNERIA



Tesi di Laurea Specialistica in

INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Design and implementation of *Prospective Search* in a real-time notification system

Candidato

Ing. Damiano Braga

Relatore

Chiar.mo Prof. Ing. Cesare Stefanelli

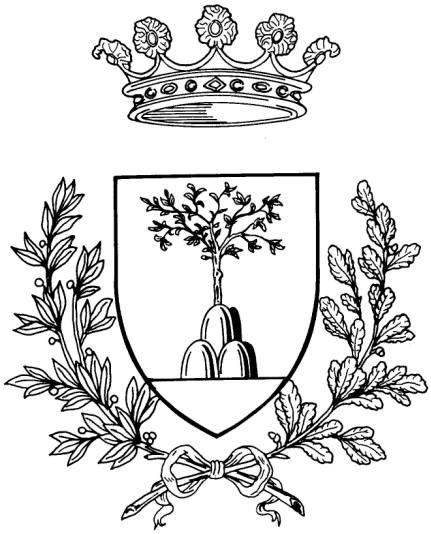
Correlatore

Ing. Daniele Farnedi

Anno Accademico 2010/2011

UNIVERSITA' DEGLI STUDI DI FERRARA

FACOLTA' DI INGEGNERIA



Tesi di Laurea Specialistica in

INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Design and implementation of *Prospective Search* in a real-time notification system

Candidato

Ing. Damiano Braga

Relatore

Chiar.mo Prof. Ing. Cesare Stefanelli

Correlatore

Ing. Daniele Farnedi

Anno Accademico 2010/2011

I do not fear computers. I fear the lack of them.
Isaac Asimov (1920 - 1992)

Alla mia famiglia,
che mi ha supportato sempre
in questi anni di duro studio.

Contents

List of Figures

Abstract

Trulia.com sends daily and weekly emails to users, notifying them about new real-estate properties. It would be more useful to send real-time notifications to the users, but this is not possible under the current system architecture in a scalable and efficient manner. A smart approach to this problem is to use a reverse search method called *prospective search*. This paper presents an implementation of a real-time notification system using the *prospective search* method on the top of the Apache Solr search platform. The result is a modular architecture where each component communicates using a Message-Oriented Middleware (MOM). Although this paper describes a project that follows certain specifications, the generality of the solution can be applied to other real-time notification systems that deal with information retrieval and publishing live on a website.

Sommario

Trulia.com invia e-mail giornaliere e settimanali agli utenti del sito web, per informarli sulla presenza di novità su proprietà immobiliari. La naturale evoluzione di questa feature sarebbe l'invio in tempo reale di notifiche agli utenti, ma questo non è possibile con l'attuale architettura in modo scalabile ed efficiente. Un approccio intelligente a questo problema è utilizzare un metodo inverso di ricerca chiamato *prospective search*. Questa tesi presenta il design e l'implementazione di un sistema di notifiche real-time che sfrutta la tecnica *prospective search* estendendo le native feature della piattaforma di ricerca Apache Solr. Il risultato è un'architettura modulare in cui ogni componente intercomunica utilizzando un Message-Oriented Middleware (MOM). Anche se questa tesi descrive un progetto che segue particolari specifiche, la generalità della soluzione può essere applicata ad altri sistemi di notifica real-time che si occupano di recupero e di invio di nuove informazioni.

Introduzione

Negli ultimi vent'anni l'esplosione di Internet ha comportato un repentino e radicale cambiamento nel mondo tecnologico. L'integrazione sempre maggiore del web nella vita di ogni giorno ha comportato un'evoluzione della modalità di creare business in ambienti prima apparentemente lontani dal mondo virtuale. Ne è un perfetto esempio il mercato immobiliare americano, dove la figura fulcro del sistema è sempre stata l'agente immobiliare. La sua funzione è di collegamento tra venditori e compratori di proprietà immobiliari. Negli ultimi anni, nonostante sia rimasto un punto fondamentale all'interno del mercato, la modalità di interazione con questa figura è fondamentalmente cambiata.

A fine marzo 2011 il 78.3% della popolazione americana utilizzava internet, evidenziando una crescita negli ultimi dieci anni del 151.7% [1]. Questa crescita esponenziale dell'integrazione di Internet nella vita di tutti i giorni ha comportato un radicale cambiamento nella modalità di agire degli utenti. Essi tendono a ricercare online le informazioni riguardo proprietà immobiliari e le compagnie che offrono il servizio di ricerca sono diventate uno step fondamentale all'inizio del processo di compravendita. Allo stesso tempo gli agenti immobiliari o brokers sfruttano le opportunità e la visibilità che Internet offre per incrementare il proprio business ed inserirsi nel processo di compravendita di loro competenza in uno stadio avanzato. A tutt'oggi Internet ha superato la figura dell'agente immobiliare come fonte attendibile ed accessibile di informazioni a riguardo di una proprietà immobiliare: circa l' 88% degli utenti che intende comprare una casa afferma di reperire informazioni preliminari online, a dispetto dell' 87% che afferma di riferirsi direttamente ad un broker come primo contatto [2].

Piú di 5 milioni di ricerche sono effettuate attraverso internet ogni mese negli Stati Uniti e l' 81% di queste attraverso l'utilizzo di motori di ricerca come Yahoo!, Google o MSN. Tuttavia i motori di ricerca generici non riescono a garantire qualità sui risultati e immediatezza se la ricerca riguarda il mondo immobiliare. Per soddisfare questo mercato compagnie che offrono motori di ricerca specifici su particolari settori si dimostrano più adatte perchè offrono una ricerca definita verticale, fornendo

informazioni molto piú strutturate e, riuscendo ad andare piú in profondità, soddisfano i criteri di ricerca degli utenti [3]. Jason Goldberg, CEO di Jobster, spiega che la ricerca definita verticale porta l'esperienza ad un passo avanti permettendo una scelta individuale, personalizzata e centralizzata su contenuti specializzati come il lavoro e il mercato immobiliare eliminando la necessità di reperire le stesse informazioni da diverse fonti [3].

É in questo scenario che si pone **Trulia.com**, un'azienda fondata nel 2006 il cui obiettivo è quello di aiutare gli utenti a reperire informazioni sul mercato immobiliare e allo stesso tempo indirizzarli verso i migliori agenti immobiliari attraverso l'utilizzo di una piattaforma web. Un fondamentale elemento che contraddistingue **Trulia** dai suoi concorrenti è la qualità delle informazioni e il modo con cui esse vengono fornite. L'esplosione dell'integrazione della tecnologia mobile ha di fatto costretto compagnie di questo tipo a fornire le stesse informazioni in un formato accessibile tramite tablet o smartphone, mantenendo peró la qualitá e la velocitá di reperimento. Il 50% degli agenti e il 47% dei brokers spende in media tra i $500 e i $2000 all'anno in tecnologia per mantenere competitivo il business. La necessità di mantenersi sempre in contatto con il cliente, e allo tempo informati sull'andamento del mercato rende necessario l'utilizzo di un laptop con connessione ad internet o complementarmente di un dispositivo mobile. Un recente sondaggio ha definito lo smartphone con accesso rapido ad internet e l'iPad come due degli strumenti ritenuti necessari da un agente immobiliare all'interno del suo lavoro [4].

Una delle richieste emergenti del mercato riguarda la possibilità di essere sempre informati a riguardo di novità riferite a classi di interesse. Solo qualche anno fa Google introdusse Google Alert, un servizio innovativo che prometteva, attraverso una semplice iscrizione, di garantire continue notifiche verso l'utente in caso di nuove informazioni. Il veicolo di trasporto di questo flusso di informazioni é naturalmente Internet, nel caso specifico l'utilizzo di Email o notifiche su dispositivo mobile. Riuscire a garantire un flusso continuo di informazioni e allo stesso tempo raggiungere utenti in qualsiasi momento e locazione sono gli obiettivi fondamentali delle maggiori aziende fruitrici di informazioni nel mondo dell'informatica attuale.

Questa tesi ha come obiettivo la realizzazione di un sistema di notifica real-time il cui scopo è quello di allertare gli utenti ogni qual volta vi siano nuove informazioni a riguardo di proprietà immobiliari. Le notifiche avvengono a seguito di una ricerca e della registrazione da parte dell'utente sul sito web **Trulia.com** e le informazioni che possono scatenare l'invio delle notifiche riguardano nuove proprietà immobiliari oppure rilevanti cambiamenti su proprietà già esistenti.

Il cuore del sistema si occupa di indicizzare le ricerche registrate dagli utenti e, ogni

qual volta vi siano nuove informazioni su proprietà immobiliari, di individuare quali ricerche risultino pertinenti alle informazioni. Per svolgere questo compito il sistema utilizza una tecnica denominata *prospective search* o reverse search. Buona parte del focus di questa tesi è sull'estendere le funzionalita della piattaforma di ricerca Apache Solr, che non supporta questa tecnica nativamente. Una volta individuato quali sono le ricerche salvate coerenti alle informazioni e relativi utenti, il sistema si occupa di inviare le notifiche agli utenti. Poichè il componente potrebbe potenzialmente generare un flusso elevato di notifiche, è richiesto un meccanismo che le collezioni e le invii a tempo prestabilito; è inoltre richiesta un'interfaccia che permetta la personalizzazione dei parametri di controllo del sistema. Le notifiche vengono inviate agli utenti tramite l'utilizzo di email.

Ogni strumento utilizzato nello sviluppo e nell'implementazione del progetto é *Open Source*. Questa scelta mirata garantisce ai componenti del sistema di ottenere un'evoluzione futura sia per caratteristiche innovative che per stabilitá, dovute al mantenimento e rilascio di nuove versioni del codice utilizzato in ogni componente da parte di una numerosa comunitá. In particolare una versione generalizzata della progettazione e dell'implementazione della *prospective search* sul prodotto Apache Solr verrá rilasciata sotto licenza *Open Source* e resa di pubblico dominio su un repository online.

Il sistema è stato realizzato a seguito di un esperienza di tirocinio a San Francisco per **Trulia.com**. L'azienda offre un servizio web consultabile da compratori, venditori e professionisti all'interno del mondo immobiliare americano.
Il Capitolo 1 presenta il progetto in generale, la situazione preesistente e i requisiti di progetto.
Il Capitolo 2 descrive invece il framework, gli strumenti e le tecnologie utilizzate in fase di design e di implementazione del sistema.
Il Capitolo 4 presenta passo passo l'implementazione, procedendo con un'analisi ad alto livello del componente fino a raggiungere dettagli implementativi e come esso si interconnetta ad altri componenti del sistema.

Introduction

In the last twenty years the explosion of Internet brought a quick and radical change in the technologic world. The major integration of the web into every day life brought an evolution of the way to create business in environments apparently far away from the virtual world. The perfect example is the american real estate market, where the main actor of the system has always been the real estate agent. Its duty is to connect vendors and buyers of real estate properties. In the last years, the real estate agent has been remained an important point in the market but the modality to interact with this figure has changed.

At the end of march 2011 the 78% of the american population used Internet, showing a growth in the last ten years of the 151.7% [1]. This exponential growth of the integration of Internet in every day life brought a radical change in the modality to interact of the users. They search online the information about the real estate properties and the companies that offer the search service have became a fundamental step at the beginning of the real estate process.

At the same time, the real estate brokers use the power and the visibility of Internet to increment their businness and insert themselves on the chain of the real estate selling process in a more advanced state. Today Internet overcomes the real estate agent as source of information regarding a real estate property: about the 88% of the user base that want to buy a house finds preliminary information about it online, versus the 87% of it that refers directly to a broker as a the first step [2].

More than 5 billion of search every month are done using Internet in the USA and the 81% are using web search engine like Yahoo!, Google or MSN. However, the generic search engines do not provide the inside scoop such as the quality and the immediacy when the content of the search is the real estate market. On the contrary more specific search engines on the market can provide more structured and deeper information because they use a search tecnhique called vertical search [3]. Jason Goldberg, CEO di Jobster, says that the vertical search bring the search experience to a further step allowing a custom, individual and centralized search on content like the job market or the real estate market helping users to get the same information

without have to get surf multiple websites [3].

In this scenario **Trulia.com**, a company founded in the 2006, helps users to gather information about the real estate market and at the same moment addresses the users to the best real estate agents on the market through a web platform. A fundamental element that distinguishes **Trulia** from its competitors is the quality and the presentation of the information. The growth of the integration of the mobile technology has forced all companies that provide these services to release out the information in a smart format in order to be accessible from tablets or smartphones while maintaing the quality and the fast access to the source. The 50% of the real estate agents and the 47% of the brokers spend on average between $500 and $2000 in a year in technology to keep the businness competitive. In order to be always connected with the client and being always informed on the market all the agents require a laptop with a fast internet connection or a mobile device. A recent survey pointed out that the smarphone and the iPad are two instruments necessary for the real estate agent job [4].

One of the emerging demands of the market refers to the ability to always be informed about news related to the classes of interest. Only a few years ago Google introduced Google Alerts, which promised an innovative service through a simple registration. The service ensures continuous notifications to the user each time new information is available. The vehicle of transport of this flow of information is of course the Internet, specifically the use of Email notifications or push notifitications trough mobile device. The main objective of the most important companies that provide information as a service is to ensure a continuous flow of information and at the same time reach users at any time and location.

The goal of this thesis is to realize a real-time notification system. Its purpose is to notify users each time new information about real estate properties are available. A user searches about its interest using the website **Trulia.com** and after a simple registration to the service, each new information triggers the notification system. The information that can trigger the send of notifications can regard new real estate properties or relevant changes on existing properties.

The heart of the system takes care of indexing all the searches stored by the user. At the same time, whenever there is new information on real estate properties, the system identifies what original saved search is relevant to the information. To perform this task, the system uses a technique called *prospective search* or reverse search. Good part of the focus of this thesis is about extending the functionality of the Apache Solr search platform, which does not natively support this technique. Once identified the saved searches that are consistent with the information and which users

are connected to the saved searches, the system takes care of sending notifications to users. The funcionality of the system suggests that it could potentially generate a high flow of notifications, therefore a mechanism that collects all the results and send the notification on a predetermined time is required. It is also necessary an interface that allows customization of the parameters of the system. Notifications are sent to users through the use of email.

Each tool used during the development and the implementation of the project is *Open Source*. This targeted choice guarantees a future evolution for each component of the system both for innovative features and for stability thanks to the strong community that maintains and releases new version of the code. In particular a generalized version of the design and the implementation of *prospective search* on Apache Solr will be relased on *Open Source* licence and published on a online repository.

The project has been realized during an intership in San Francisco California at **Trulia.com**. The company offers a web service accessible from buyers, sellers and professionists in the american real estate market.

Chapter 1 presents the project in general, the current situation and the project requirements.

Chapter 2 describes the framework, the tools and the technologies used in the design and implementation phase of the project.

Chapter 4 describes the implementation starting with an high level description of the component until reaching implementation details and the integration in the system.

Chapter 1

Background and Analysis

The project stems from the necessity to create a real-time notification system that can be used to notify users when new information comes available.

This chapters describes the background of the project: the current situation and the necessary integration with legacy applications. In particular the attention is placed on the weakness of the architecture and if the current component could be able to handle real-time notifications.

1.1 Current situation

1.1.1 Current System

Trulia offers on the website a feature called *save this search* (Figure 1.3). This characteristic provides the ability to save a previous search on the website and receive notifications via email when a new result matches the saved search. The User can choose the frequency of the notifications between daily and weekly.

Figure 1.1 shows a diagram of the feature: the User goes to the website and search about a real estate property. After, he decides to save the previous search because he is interested to receive notification about the search if new information comes available.

The current system that handles this feature is shown both in Figure 1.4 and in Figure 1.5. All of the User's saved searches are stored into MySQL database tables. The most important table is called *UserQuery* and stores important information about the search such as the timestamp, the unique User identification and the unique search identification. The information about the real estate properties are indexed into an Apache Solr index cluster. During night time, a batch process responsable to handle the *save this search* feature, takes all the information about saved search

stored in the MySQL tables. This component takes each saved search and the relate query used originally to execute the seach and it executes the same query again against the Apache Solr index cluster. The results of the search on the cluster is processed and if contains new information the User connected to that search has to be notified. At the end of the batch process a result collector posseses all the new real estate property information and the Users information. At this point an email notifier takes all the information from the result collector and sends out the email to the Users. Prior sending out the notifications, the notifier has to check the User preferences about the frequency of the notification. Infact the User decides between daily and weekly notifications. In the second case the notifier has to consider all the updates about the past week.

1.1.2 Weaknesses

The current system has been designed to support daily and weekly notifications and it originally worked with a small amount of Users and saved searches. For this reason the growth of the user base showed few glitches in the architecture of the component expecially on the scalability. Plus the current system can not be used to handle real-time notifications for these reasons that are connected directly with the architecture of the component:

1.1.2.1 Large load on Search Servers

As already seen, the whole set of saved search are stored into MySQL database tables. The batch process executes each saved query against the search server cluster. Even if the process is executed during the night to avoid interferences between the bot traffic and the user traffic to the search servers, the number of saved searches is considerable and consequently the load on servers. Figure 3.2 and 3.3 shows the load on a random search server during night. The performance degradation experienced during this period should be imagined continuous in case of real-time notification if the current component is used.

1.1.2.2 Scalability

The system shows difficulty handling the current user base community and related saved searches and that's one of the other reasons why the batch process runs during the night. Even if the current system could be able to handle real-time notifications, the present approach is not scalable in the case of an increase of the user base of the website. The growth of the number of users could potentially bring a large number

of new saved searches and hereafter a growth of the load on search servers until the entire system will not be able to handle the entire process. In this case trying to scale the system both vertically or horizontally would not bring great advantages because all the set of saved search will be executed against the search servers cluster as shown in Figure 1.2 and both the database that contains the saved searches and the search servers are a performance bottleneck.

1.1.2.3 Effectiveness

The major weakness of this architecture is the big load caused by the execution of the saved search on the search servers cluster. The entire purpose of this action is to find all the new information or updates on the saved searches. However, this all approach is not effective assuming that less than the 10% of the data changes each day. It would be preferable if the component would use the resources only when necessary, thereby reducing wastage and speeding the process itself.

1.1.2.4 Performances

The results collector which collect all the updates and the new information from the results of the execution of the saved searches has to wait the end of the process each time the batch process runs. That does not represent a problem if the process fetches daily and weekly updates but is relevant if the system has to handle real-time updates. In this case, the first notification would be sent at the end of the overall process. This time frame is directly connected to the number of saved searches and to the performances of the search servers cluster.

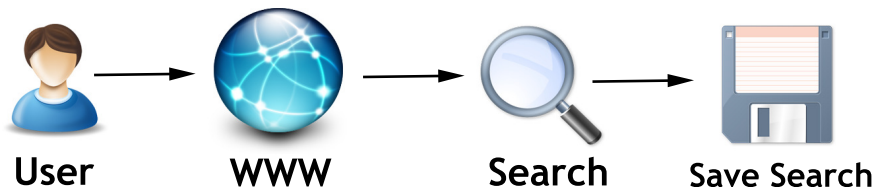


Figure 1.1: *The user search about real estate properties and then saves the search.*

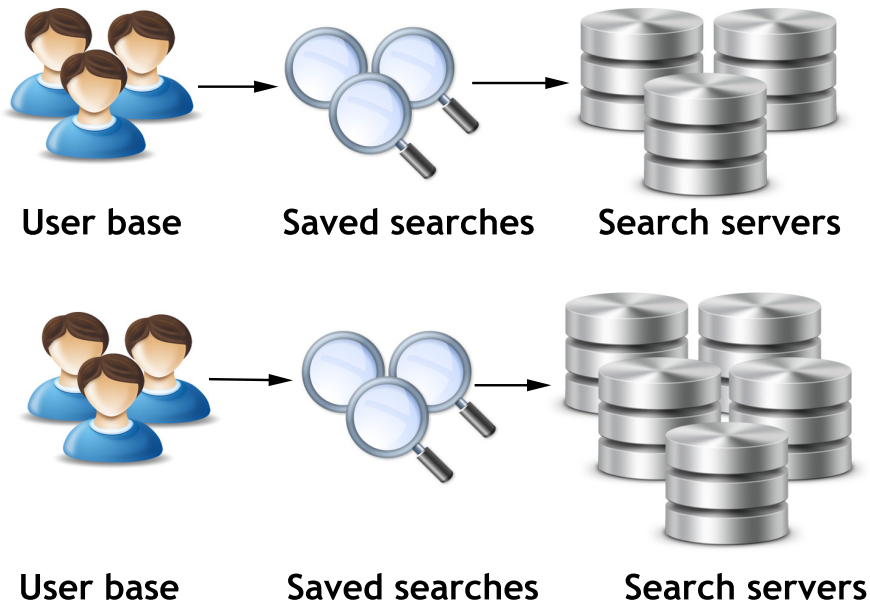


Figure 1.2: *Scaling up and scaling does not help to improve performance a lot*

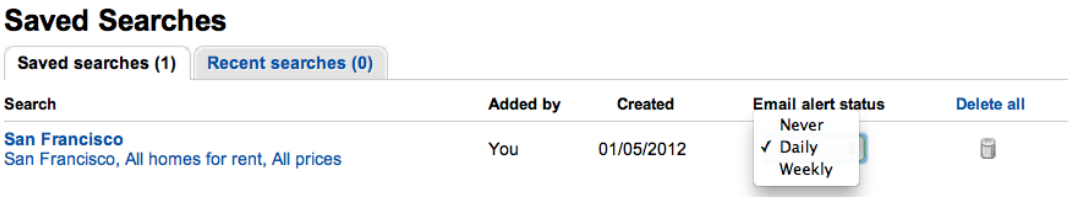


Figure 1.3: *Recap window of saved search from the user*

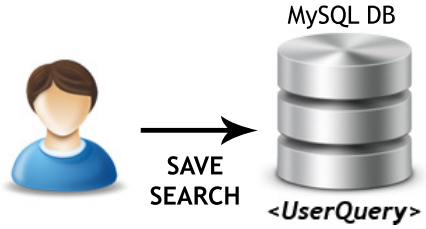


Figure 1.4: *Previous situation : User saves a search*

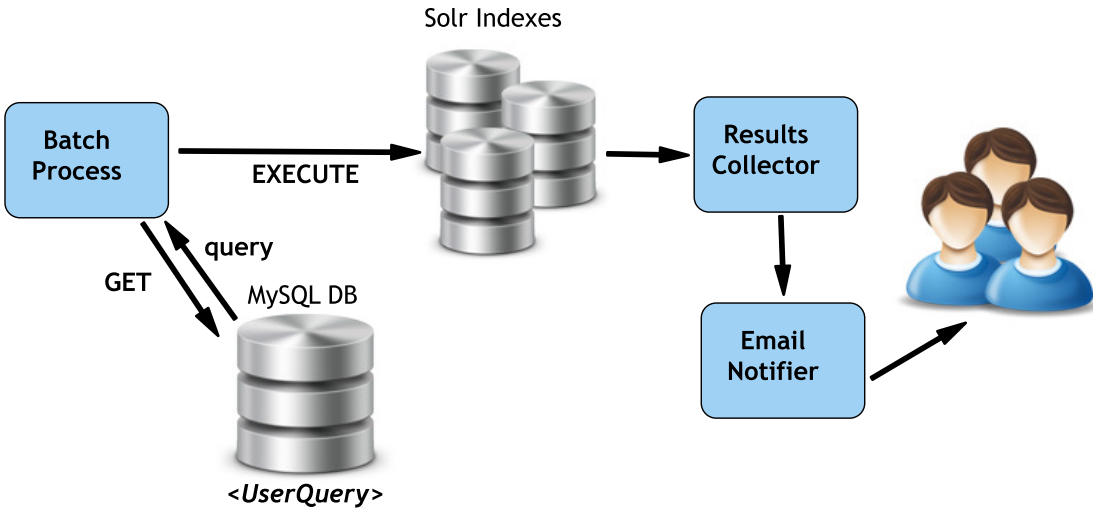


Figure 1.5: *Previous situation : Daily and Weekly notification system*

Chapter 2

Technologies and tools

This project combines the use of different feature and tools, each one is responsible for handling a particular funcionality of the application.

This chapters describes the technologies and the tools used during the project. Each element has been chosen after a targeted study on the performance and the necessary integration with the system. The Spring Framework is used to speed the development allowing reusability but at the same time to use conventions to help future development. The application run inside an application container called Jetty to allow an easy control of the life cycle and a easy maintenance. Since the project is a mixture of different technologies, Maven is used as a project managment tool to describe and help the development process. Apache Solr is a search platform that can store, index and make information searchable. The communication between the various elements of the application is made with ActiveMQ, a messaging system that uses the JMS APIs and also implements the publisher/subscriber model.

2.1 Framework

2.1.1 Spring Framework

Spring Framework is a open source Java platform that provides an infrastructure support for developing Java applications [8]. The main objective of Spring is to handle the complexity of the development of enterprise applications in a Java environment. The Spring Framework allows to build application from *plain old java objects* (POJOs). POJO is a Java object that doesn't extend or implement some specialized classes and interfaces respectively require by the Enterprise Java Bean technology. Spring apply enterprise services to POJOs allowing to become a valid substitute to the old Java Beans technology or to the EJB (Enterprise Java Beans) classic architecture.

The Java Beans technology provides simple java classes (beans) with defined nomenclature rules to access: properties are available thanks to getter and setter methods. This technology is still used mainly while dealing with visual components in the user interface but is not used while building enterprise applications due the semplicity of the components. Moving forward, the Enterprise JavaBeans (EJB) technology provides a standard way to implement the business code typical of the enterprise applications. Most of the common enterprise problems can be solved using EJB that provides all the system services to the java beans thanks to an application server. The relation between beans and the overall system is made with contracts implemented with particular interfaces and providing services declared with specific XML configuration files called deployment descriptors. However, even if EJB brought evolution on the components, it also brought an unnecessary complexity in the development of enterprise applications. The forced use of the interfaces makes the EJB code invasive and the development is slowed by the necessity to write XML deployment descriptors. Spring tries to overcome this limit providing two new models of programming: Aspect Oriented Programming (AOP) and Inversion of Control (IoC). This techniques add a declarative model on the top of the java beans like the EJB technology, without the complexity. The programmer can focus on the logic aspects of the application without being overwhelmed by the complexity of writing a EJB component.

2.1.1.1 Inversion of Control

Inversion of Control (IoC) is software design pattern and a set of associated programming techniques in which the flow of control of a system is inverted in comparison to the traditional interaction mode. From the Spring prospective the IoC is often referred as Depedency Injection (DI). This feature focuses on injection of dependencies (required resources) into the dependent components at run time. In the java beans technology, a bean controls the instantiation or the location of its dependencies by using the direct construction of classes or a Service Locator pattern. The opposite is the Spring IoC, hence the name *inversion*, where objects define their dependencies and the container injects them when it creates the bean. There are three main ways through which dependencies are injected into the dependent components.

- **Constructor Injection:** the dependencies are injected using a constructor and so the constructor needs to have all the dependencies or the references declared into it.

- **Setter Injection:** this form uses the setter methods to inject the dependencies into the dependent components. All the dependent objects must have setter methods in their respective classes which would eventually be used by the Spring IoC container.

- **Interface Injection:** any implementation of the interface can be injected. The container uses the declared injection interfaces to figure out the dependencies and the injectors to inject the correct dependents.

The IoC Container implements of the IoC and the DI on the Spring Framework. The container manages the objects that form the backbone of the applications. The dependencies between them are reflected in the configuration metadata used by the container. The actual representation of the Spring IoC container is the `org.springframework.beans.factory.BeanFactory` interface, and it's responsible for containing and managing the beans. Figure 2.1 shows what is the relation between a POJO and the Spring container.

The BeanFactory interface is responsible for instantiating or sourcing application objects, configuring such objects, and assembling the dependencies between these objects. The BeanFactory applies the Inversion of Control (IOC) pattern to separate an application's configuration and dependency specification from the actual application code.

The most commonly used BeanFactory implementation is the XmlBeanFactory class. This implementation allows to express the objects that compose the application, and the interdependencies between such objects, in terms of XML code. The XmlBeanFactory takes this XML configuration metadata and uses it to create a fully configured system or application.

2.1.1.2 Modules

One of the main advantages of the Spring framework is its layered architecture, which allows to be selective about which of its components can be used providing a consistent framework for J2EE application development. The modules can be grouped in macro groups as shown in Figure 2.2. The Spring modules are built on top of the core container, which defines how beans are created, configured, and managed.

The *Core Container* provides the fundamental functionality of the Spring framework. In this module primary component is the BeanFactory.

The *Spring context* is a configuration file that provides context information to the Spring framework. The Spring context includes enterprise services such as email,

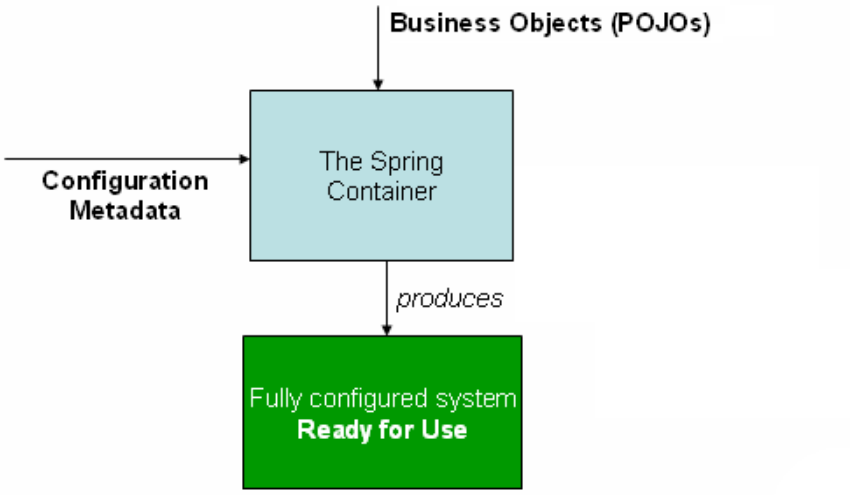


Figure 2.1: *Spring Framework: IoC container*

JNDI, EJB, internalization, validation, scheduling and applications lifecycle events. Also includes the support for the integration with templating frameworks.

The *Spring AOP* module allows a software component to be decorated with additional behavior, through its configuration management feature. It is possibile to AOP-enable any object managed by the Spring framework. The AOP module provides transaction management services for objectsand it is possible to incorporate declarative transaction management into your applications without relying on EJB components.

The *Spring DAO* module provides a JDBC-abstraction layer that reduces the need to do JDBC coding and parsing of database-vendor specific error codes. The JDBC package provides also a way to do programmatic and declarative transaction management for all POJOs.

Spring provides integration with OR mapping tools like Hibernate, JDO and iBATIS. Spring transaction management supports each of these *ORM* frameworks as well as JDBC.

The *Web context* module provides basic web-oriented integration features builds on top of the application context module, providing contexts for Web-based applications. The Spring framework supports integration with Jakarta Struts. The Web module also eases the tasks of handling multi-part requests and binding request

Spring provides a pluggable MVC architecture. The users have a choice to use the web framework or continue to use their existing web framework. Spring separates the roles of the controller; the model object, the dispatcher and the handler object which makes it easier to customize. Spring web framework is view agnostic and does

not push the user to use only JSPs for the view. The user has the flexibility to use JSPs, XSLT, velocity templates etc to provide the view.

The Test module contains the *Test* Framework that supports testing Spring components using JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also contains a number of Mock objects useful in many testing scenarios.

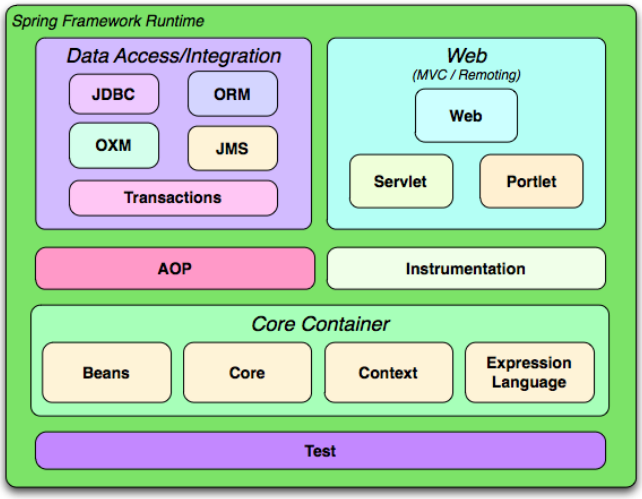


Figure 2.2: *Spring Framework: modules*

2.1.1.3 Life cycle of a Spring Bean

As already said, all Spring Beans reside within a Spring Container also known as IOC Container. The Spring Framework is transparent and thereby hides most of the complex infrastructure and the communication that happens between the Spring Container and the Spring Beans. A bean goes through various stages during its lifetime, or life cycle. Spring Beans exist within the Container as long as they are needed by the appplication. There are various lifecycle interfaces and methods that are called by the IOC Container.

Firstly, the Bean Container finds the definition of the Spring Bean in the Configuration file. At this point the Bean is recognized and the Container creates an instance of the Bean using Java Reflection API. Any properties mentioned along with the Bean are applied and if the property itself is a Bean, then it is resolved and set. Before the initialization phase, the container checks if `BeanNameAware` and `BeanClassLoaderAware` interfaces are implemented by the Bean and then procedes calling `setBeanName()` and `setBeanClassLoader()` methods passing the name of the Bean and an instance of the ClassLoader object that loaded the Bean. If there

are any post processor objects associated with the BeanFactory that loaded the Bean, the method `postProcessBeforeInitialization()` is called and then the container start to inizialize the Bean. If the Bean class implements the `InitializingBean` interface, then the method `afterPropertiesSet()` and all the Bean properties defined in the Configuration file are set. If the Bean definition in the Configuration File contains a init-method attribute, the value of the attribute is resolved to a method name in the Bean class and called. In order to conclude the lifecyle of the Bean, if the original Bean class implemented the `DisposableBean` interface, then the method `destroy()` will be called when the application no longer needs the bean reference. If the Bean definition in the Configuration file contains a destroy-method attribute, then the corresponding method definition in the Bean class will be called.

2.1.2 Jetty

Jetty is a Java-based HTTP open source client and server, a WebSocket client and server and also an application server. It is developed as a free and open source project as part of the Eclipse Foundation [9]. As shown in Figure 2.3, the Jetty Server is between a collection of Connectors that accept HTTP connections, and a collection of Handlers that service requests from the connections and produce responses. The process is done by threads taken from a thread pool.

Jetty can be invoked and installed as a stand alone application server or it can be easy embedded in an application or a framework as a HTTP component, as a simple servlet engine or as a part of a full JEE envirnoment.

In order to configure Jetty, a network of connectors and handlers must be built and each one has to have its own configurations. Each components of Jetty is a Plain Old Java Objects (POJOs) and the configuration can be done using simple XML files or using Dependency Injection with the help of the Spring Framework.

The modular nature of Jetty allows deployments to be targeted at precisely the set of services required without the additional complexities, inefficiencies and security concerns of unused features.

2.1.2.1 Jetty and the Spring Framewok

The configuration of Jetty can be done with almost any IoC style framework including Spring. The Jetty-Spring module is included in `Jetty-hightide` package and the Spring dependencies can be downloaded and installed into `$JETTY_HOME/lib/spring`. The Jetty configuration can be done inside the Spring framework calling the API as Spring beans. The main configuration file is called `jetty-spring.xml` and it must

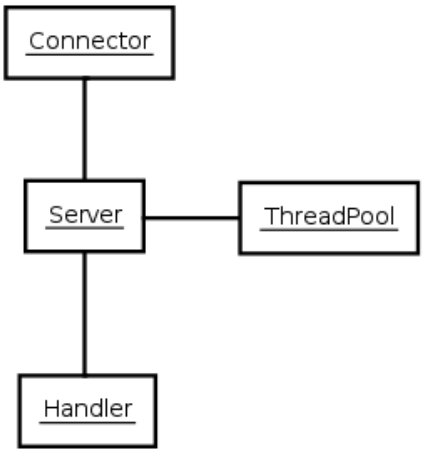


Figure 2.3: *Jetty: Architecture*

define the threadPool, the connectors and the handlers. Finally the Jetty container can be started normally utilizing the `start.jar` jar file inside the `$JETTY_HOME` main directory.

2.1.3 Maven

Apache Maven is an open source project management tool which encompasses a project object model, a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle [10]. The project is described using a well-defined project object model (POM) and Maven can apply cross-cutting logic from a set of shared (or custom) plugins. Maven uses the CoC (Convention over configuration) concept that is systems, libraries, and frameworks should assume reasonable defaults without requiring unnecessary configuration. Maven incorporates this concept by providing sensible default behavior for projects. Without customization, source code is assumed to be in `$basedir/src/main/java` and resources are assumed to be in `$basedir/src/main/resources`. Tests are assumed to be in `$basedir/src/test`, and a project is assumed to produce a JAR or a WAR file.

As already mentioned Maven has been designed to delegate most responsibility to a set of Maven Plugins which can affect the Maven Lifecycle and offer access to goals. Most of the action in Maven happens in plugin goals which take care of things like compiling source, packaging bytecode, publishing sites, and any other task which need to happen in a build. Most of the intelligence of Maven is implemented in the

plugins and the plugins are retrieved from the Maven Repository. This is behavior which allows to upgrade a plugin to add capability to the project's build. The fact that Maven retrieves both dependencies and plugins from the remote repository allows for universal reuse of build logic.

Maven uses a construct known as a Project Object Model (POM) to describe the software project being built, its dependencies on other external modules and components, and the build order. It comes with predefined targets for performing certain well-defined tasks such as compilation of code and its packaging. In order to use maven to build a project the source code should be placed in `$basedir/src/main/java`. After running the simple command `mvn install` from the command line, Maven will process resources, compile source, execute unit tests, create a JAR file or a WAR file and install the JAR in a local repository for reuse in other projects.

Maven allows to install and use Spring framework declaring the dependencies with a simple configuration in the POM file as shown in the fragment 2.1 .

Listing 2.1: *Fragment of the Spring Framework configuration in the POM maven file*

```
<dependencies>
    ...
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.5.6</version>
    </dependency>
  </dependencies>
```

2.2 Messaging

2.2.1 Publish/Subscribe Pattern

Publish/Subscribe messaging (P/S) is a model in which the sender of the message, known as a *publisher*, does not know directly to who send the message. Similary the receiver, known as a *subscriber*, does not know who is the publisher. The key role in this model is represented by an entity in the middle that should automatically forward all the publisher's messages to the right subscriber thanks to a set of rules that usually a subscriber or a publisher decide. Messages are divided into classes according to their type. Each subscriber decides in which classes wants to subscribe in order to receive notifications.

The process of routing the messages to the right subscriber is also called filtering. There are many message filtering methods in the Publish Subscribe model, but the

most important are :

- **Topic-based**: publisher sends messages to a *topic* or a logic channel. The subscriber will subscribe to topics of its interest and it will receive a copy of the messages sent to that channel. The publisher is responsible for deciding which classes of messages are sent to the topic. A diagram is shown in Figure 2.4. In this scenario, the same message could be sent to multiple subscribers if they all have the same type of subscription.

- **Content-based systems**: a message is sent to a subscriber if only the attributes or the *content* of the message meets the requirements decided by the subscriber. That means that the subscriber is responsible for classifying the messages. A diagram is shown in Figure 2.5. The message broker is represented by a queue that is used by the publishers to send messages. The messages are received by the subscribers.The simple version of this scenario is rappresented by a single subscriber per each queue.

- **Hybrid systems**: There are also other systems where the logic of the filtering does not reside inside the subscriber or the publisher but in the message broker itself. Publisher send messages to the broker like in the topic-based system. Subscribers register subscribtions with that broker, letting the broker perform the filtering. Since the filltering logic is inside the message broker, the number of subscribers waiting for messager by the broker can be from one to many.

  P/S has a number of interesting characteristics. Firstly, this system is loosely coupled: publishers do not need to address subscribers and vice versa. Instead, subscribers simply specify the messages they are interested in. Secondly, communication is asynchronous. Thirdly, publishers and subscribers do not need to be available at the same time: if a publisher join the system after a subscribtion was issued, that subscription triggers the notifications.

2.2.2 JMS

Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between clients. JMS is a part of the Java Platform, Enterprise Edition, and it is defined by a specification developed under the Java Community Process as JSR 914 [5].

The Java Message Service API allows applications to create, send, receive, and read messages and defines a common set of interfaces and associated semantics that

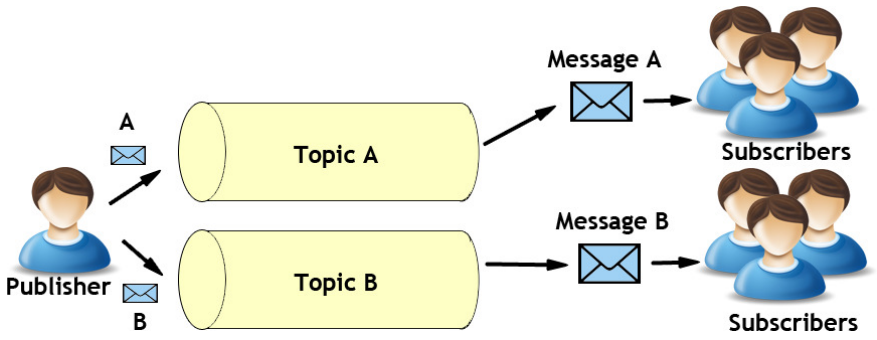


Figure 2.4: *Publish/Subscribe: topic-based message filtering*

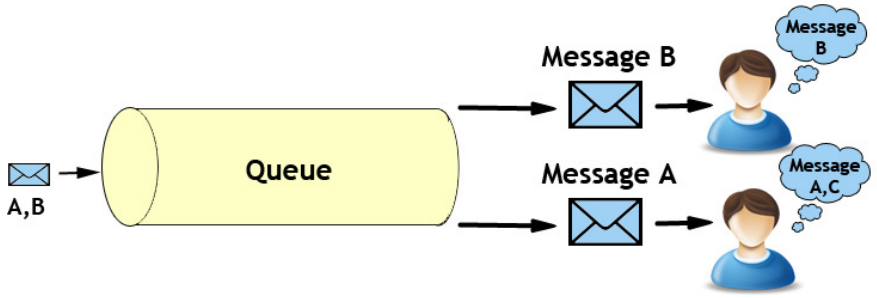


Figure 2.5: *Publish/Subscribe: content-based message filtering*

allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn to use messaging products but provides enough features to support sophisticated messaging applications. It also maximize the portability of JMS applications across JMS providers in the same messaging domain.

The JMS API enables communication that is not only loosely coupled but also asynchronous. That means that a JMS provider can deliver messages to a client as they arrive and a client does not have to request messages in order to receive them. At the same time the communication is reliable: the JMS API can ensure that a message is delivered once and only once. A JSM application is generally composed by:

- **Provider:** The JMS provider is the vendor-specific MOM that implements the JMS API. Such an implementation provides access to the MOM via the standardized JMS API.

- **Client:** The program or component, written in the Java programming language, that produces and consumes messages. JMS clients utilize the JMS

API for interacting with the JMS provider. They use the API for standardized access to the messaging service. Many JMS provider implementations (including ActiveMQ) include features beyond those required by JMS. JMS clients utilize the `MessageProducer` and `MessageConsumer` interfaces in order to send and receive messages. The JMS provider has the responsability to furnish the implementaion of these interfaces. A JMS client that sends messages is known as a *producer* and a JMS client that receives messages is known as a *consumer*. It's possible for a JMS client to handle both the sending and receiving of messages. The JMS `MessageProducer` class is used to send messages to a destination. The default destination for a given producer is set when the producer is created using the `Session.createProducer()` method. The `MessageProducer` provides methods not only for sending messages but also for setting various message headers including the `JMSDeliveryMode, the JMSPriority, the JMSExpiration`, as well as a utility `send()` method for setting all three of these at once. The JMS `MessageConsumer` class is used to consume messages from a destination. The `MessageConsumer` can consume messages either synchronously by using one of the `receive()` method or asynchronously by providing a `MessageListener` implementation to the consumer.

- **Messages:** the objects that communicate information between JMS clients. The JMS message is the most important concept in the JMS specification. A JMS message allows everything to be sent as payload of the message, including text and binary data as well as information in the headers. As shown in Figure 2.7, JMS messages contain two parts, including headers and a payload. The header provides metadata about the message used by both clients and JMS providers. The payload is the actual body of the message and can hold both textual and binary data via the various message types. The JMS message is designed to be easy to understand and flexible. All the complexity of the JMS message resides in the header. The JMS API defines five message body formats, also called message types (shown in Table 2.1) , which allow to send and to receive data in many different forms and provide compatibility with existing messaging formats.

- **Administered objects:** Administered objects contain provider-specific JMS configuration information and are supposed to be created by a JMS administrator. Administered objects are used by JMS clients. They're used to hide provider-specific details from the clients and to abstract the JMS provider's

administration tasks. It's common to look up administered objects via JNDI but it's not required. The JMS spec defines two types of administered objects: `ConnectionFactory` and `Destination`.

- **Native clients:** programs that use a messaging product's native client API instead of the JMS API

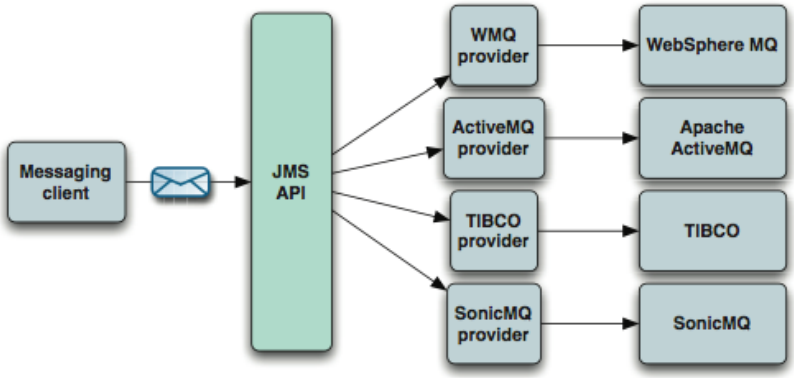


Figure 2.6: *JMS allows a single client to easily connect to many JMS providers.*

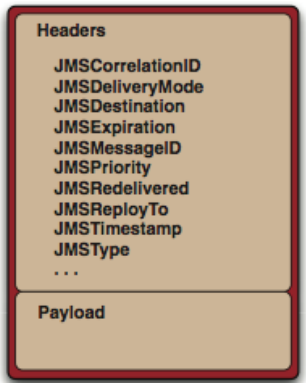


Figure 2.7: *JMS message*

Figure 2.8 shows the components of the JMS API and how the interacts. Administrative tools allows to bind destination and connection factories into a Java Naming and Directory Interface (JINDI) API namespace. A JMS client can then look up the administered objects in the namespace and then establish a logical connection to the same objects throught the JMS provider.

The JMS Specification explains that the messages can be consumed in either synchronously or asynchronously. In the first case, the receiver of the message

explicty fetches the message from the destination by calling a blocking method. In the second case, the client can use a message listener with a consumer. Whenever a message arrives to the destination, the JMS provider delivers the message notifying the listener.

The basics building blocks of a JMS application are shown in Figure 2.9.

2.2.2.1 Connection Factory

JMS clients use the `ConnectionFactory` object to create connections to a JMS provider. Connections typically represent an open TCP socket between a client and the JMS provider, so the overhead for a connection is large. While dealing with this kind of connection, the use of a pool of connections shared between the application is recomended. JMS connections are used by JMS clients to create `javax.jms.Session` objects that represent an interaction with the JMS provider. Each time an application complete its work flow the connection must be closed or else the resources can not be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers. Each connection, after being instantiated must be started with the start method. There is also a stop method that can temporary stop the message delivery withouth losing the connection to the provider.

2.2.2.2 Destination

The Destination object encapsulates the provider-specific address to which messages are sent and from which messages are consumed. Although destinations are created using the Session object, their lifetime matches the connection from which the session was created. Temporary destinations are unique to the connection that was used to create them. They'll only live as long as the connection that created them and only the connection that created them can create consumers for them.

2.2.2.3 Session

A session is a single-threaded context for producing and consuming messages. The sessions can be used to create message producers, message consumers, and messages. Sessions serialize the execution of message listeners.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

Message Type	Body Contains
TextMessage	A java.lang.String object
MapMessage	A set of name/value pairs, with names as String objects and values as primitive types in the Java programming language.
BytesMessage	A stream of uninterpreted bytes
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Composed of header fields and properties only

Table 2.1: *JMS Message Types*

2.2.2.4 Message producer

A message producer is an object created by a session and is used for sending messages to a destination.

2.2.2.5 Message consumer

A message consumer is an object created by a session and is used for receiving messages sent to a destination. A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

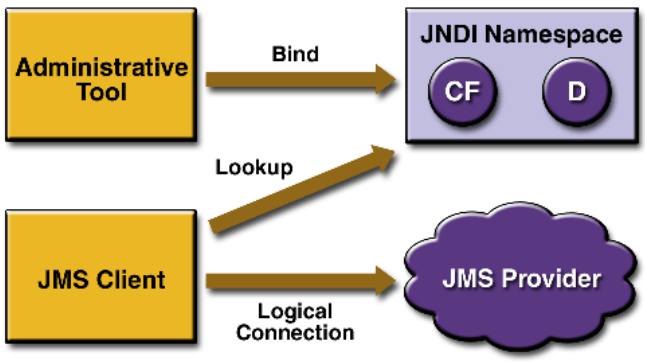


Figure 2.8: *JMS API architecture*

2.2.3 Apache ActiveMQ

ActiveMQ is an open source MOM system that is developed and maintained by the Apache Software Foundation community. It's a message brokers which fully

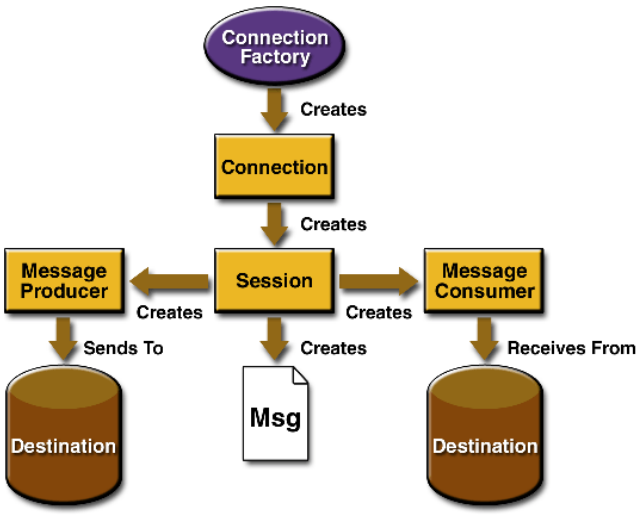


Figure 2.9: *Building blocks of a JMS application*

implements the JMS API and provides quality of service (QoS) features such as performance, scalability, availability, reliability and security. Even if is fully written in Java, with the primary goal of implementing the JMS API, ActiveMQ also supports other programing languages such as C#, Ruby, Python, C/C++. The message broker supports a myriad of different low-level transport protocol such as TCP, SSL, HTTP, HTTPS, and XMPP and its modular architecture makes it an extendible messaging framework.

The main features that ActiveMQ guarantees are:

- **Availability:** a completely transparent load balacing, and the implementation of a failover and a reliable protocol that allows to automatically reconnect in case of a fail and it will also resume any temporary destinations, sessions, producers and most importantly consumers.
- **Interoperability:** between various message stores including JDBC
- **Scalability:** use of clustering and load balacing
- **JMS compliance:** ActiveMQ is an implementation of the JMS 1.1 spec. This provides important benefits and guarantees, including synchronous or asynchronous message delivery, once-and-only-once message delivery, message durability for subscribers etc.
- **Persistance and security:** ActiveMQ provides persistance in many forms, also security can be completely customized for the type of authentication and

authorization that's best for the application. The default authorization is made using properties files as well as standard JAAS login modules.

- **Integration with application servers:** it is possible to integrate ActiveMQ with a Java application server, including Tomcat, Jetty, Apache Geronimo, and JBoss.

- **Simple administration:** It does not require a dedicated administrator because it provides administrator features. The tool can be administered using JMX, or JConsole or also using the web console provided by ActiveMQ.

2.2.3.1 Components

The more important components of the ActiveMQ messaging framework are the client, message, destination, and the message broker. The interaction between the components is shown in Figure 2.10. The client is the application component that use the service provided by the message broker and can be further categorized as either a message *producer* or *consumer*. The role of the producer is to create the message, which is assigned to the message broker for routing and delivery to a particular destination. Consumers retrieve messages from the destinations which they have been routed. A destination can be seen as a logical channel through which clients communicate with one another. The broker has the role to route the messages to the correct destination and also to ensure the right QoS. The message broker can be also considered as a cluster of brokers in case the system is complex. Figure 2.11 shows that a destination can be either a *queue* or a *topic* and is maintaned by the message broker. Queues are used to the broker to send a message to a consumer using 1:1 relationship, while topics are used by a producer to send a message to one or more consumers using 1:N relationship. There are two basics types of ActiveMQ brokers:

- **Embedded:** it's executed within the same JVM process as the Java clients that are using its services. One ore more client can reside in the single JVM, each executing within its own thread of execution. The clients communicate with their embedded broker via direct method invocation (DMI) as opposed to serializing command objects (messages) across a TCP/IP-based transport connector. If the network fails, its embedded clients can still use the services of the broker. Performance are also increased using the embedded broker due to the use of the DMI.

- **Standalone**: it does not have its clients residing in its JVM and communicates with its clients via network-based transport connectors. The connectors rapre-

sent network communication channels through which the clients communicate with their respective brokers and brokers communicate with one another; a list of the most important connectors are shown in Table 2.2 but the most used is defenitely OpenWire.

2.2.3.2 Failover Protocol

Normally a client can be configured to connect to one specific broker; in case of connection problems at the first stage the connection is not initiated. At the later stage, if some connection problem happens the client can chose to stop the normal workflow or else try to restablish the connection. That's where protocols such as failover come in to play. It implements automatic reconnection. A default configuration implements reconnection delay logic, meaning that the transport will start with a 10ms delay for the first reconnection attempt and double this time for any subsequent attempt up to 30000ms. Also, the reconnection logic will try to reconnect indefinitely. All reconnection parameters can be reconfigured. The advantage of this feature is that clients don't need to be manually restarted in the case of a broker failure (or maintenance). As soon as the broker becomes available again the client will automatically reconnect. This means far more robustness for the applications. The failover transport connector plays an important role in achieving advanced functionalities such as high availability and load balancing.

2.2.3.3 OpenWire Protocol

OpenWire is a cross language wire protocol that allows native access to ActiveMQ from a number of different languages and platforms. The Java OpenWire transport is the default transport from the version 4.x of ActiveMQ. OpenWire is used to marshal objects to byte arrays and back. Each marshaled objects can be referred as a command. A TCP network connection will se multiple commands back to back on the stream of the communication. Commands are not delimited in anyway and they can have variable size. All the primitive data type that are used in the commands are encoded in big-endian byte order.

Even if OpenWire is extensible and supports adding new encoding options, each OpenWire protocol session starts with all encoding options turned off. An initial `WIREFORMAT_INFO` command is exchanged between the two nodes so that additional encoding features can be enabled as shown in Figure 2.12. Only if both side of the communication request an encoding feature to be enabled it will be enabled. Every command has three fields:

- **Size:** number of bytes that are present in the command
- **Type:** a command type identifier
- **Command specific fields:** contains all the data for the command. The encoding depends from the type used.

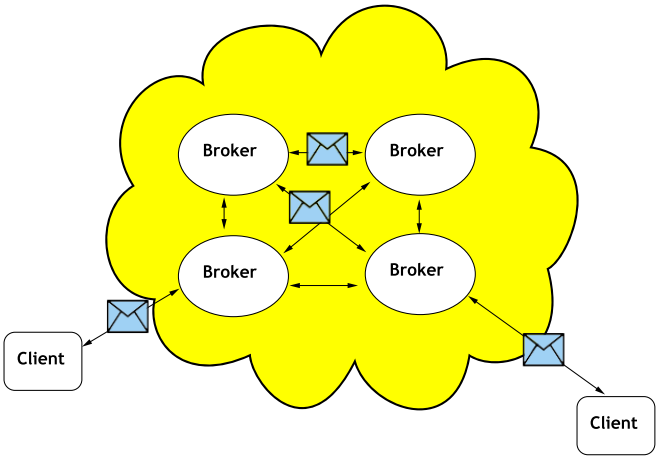


Figure 2.10: *ActiveMQ components*

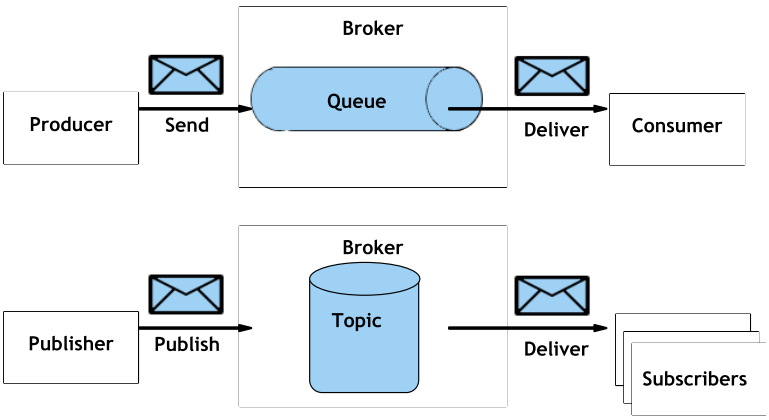


Figure 2.11: *ActiveMQ: Queue and Topic*

Wire Protocol
OpenWire
Stomp
REST
XMPP

Table 2.2: *ActiveMQ connectors*

Figure 2.12: *OpenWire; Wire Format Negotiation*

2.3 Logging

2.3.1 Apache Log4j

Each enterprise appplication needs a logging system, both for debugging purposes and for monitoring the normal life cycle of the application. The Spring Framework allows the integration with differents logging system, but one of the most used is the open source product *Apache Log4j*. Log4j is a popular logging package written in Java. One of its distinctive features is the notion of inheritance in loggers. Using a logger hierarchy it is possible to control which log statements are output at arbitrary granularity. This helps to reduce the volume of logged output and minimize the cost of logging.

One of the advantages of Log4j API is its manageability. Once the log statements have been inserted into the code, they can be controlled via configuration files. They can be selectively enabled or disabled, and sent to different and multiple output targets in user-chosen formats. The log4j package is designed so that log statements can remain in shipped code without incurring a heavy performance cost.

Log4j has three main components: loggers, appenders and layouts. These three types of components work together to enable developers to log messages according to message type and level, and to control at runtime how these messages are formatted and where they are reported. The first and foremost advantage of any logging API resides in its ability to disable certain log statements while allowing others to print unhindered. This capability assumes that the logging space, that is, the space of all possible logging statements, is categorized according to some criteria.

Loggers are named entities and the names are really important to the logic of the system. A rule relates loggers on the hierarchical criteria: a logger is said to be an

anchestor of another logger if its name followed by a dot is a prefix of the descendant logger name. A logger is said to be a parent of a child logger if there are no ancestors between itself and the descendant logger. Loggers may be assigned levels. The set of possible levels in order of importance:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL

If a given logger is not assigned a level, then it inherits one from its closest ancestor with an assigned level. Configuration of the Log4j environment is typically done at application initialization, but the environment could be changed on runtime, allowing the application to switch to debugging mode.

Log4j allows logging requests to print to multiple destinations: each destination is called appender. Currently, appenders exist for the console, files, GUI components, remote socket servers, JMS, NT Event Loggers, and remote UNIX Syslog daemons. It is also possible to log asynchronously. Figure 2.13 shows the components of Log4j.

2.4 Indexing

2.4.1 Apache Solr

Apache Solr is an open source enterprise search platform based on a Lucene core. It's entirely written in Java but it supports a variety of languages such as Python, PHP, Ruby thanks to the extend usage of JSON APIs. It is a product that power search for public sites like Ebay, Zappos, Netflix, AOL and it is considered to be a mature project even if he was donated to the Apache Software Foundation in the 2006.

The standard feature of Apache Solr is to return a list of search results for a specific query. However, the product offers numerous other features such as hit highlighting, faceted navigation, query spell correction, dynamic clustering, database integration, rich document (e.g., Word, PDF) handling. The last stable version of

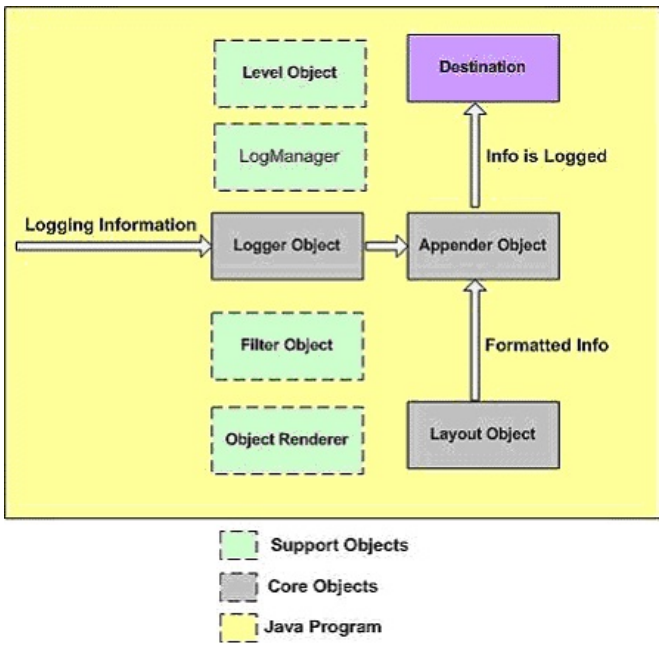


Figure 2.13: *Logging: Log4j components*

Apache Solr supports also the so called geospatial search. It provides distributed search and index replication [6].

It usually runs inside a servlet container like Apache Tomcat [7] or Jetty [9]. Apache Solr is a really flexible product: it could be configured to work with pretty much any application and adds a level of customization thanks to a powerful plugin architecture.

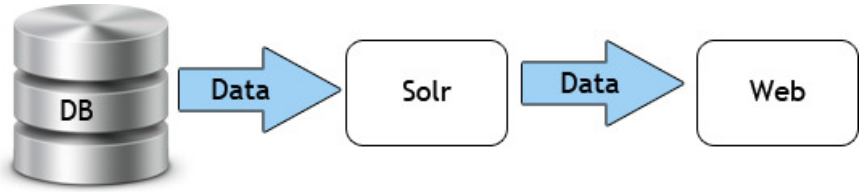


Figure 2.14: *Apache Solr: common usage*

2.4.1.1 The document

The basic unit of information is the *document*, which is a set of data that describes something. The structure of a document depends mostly from the structure of the information stored. Each document is composed by *fields*, which are most specific pieces of information. There are different types of fields, each one is treated in a

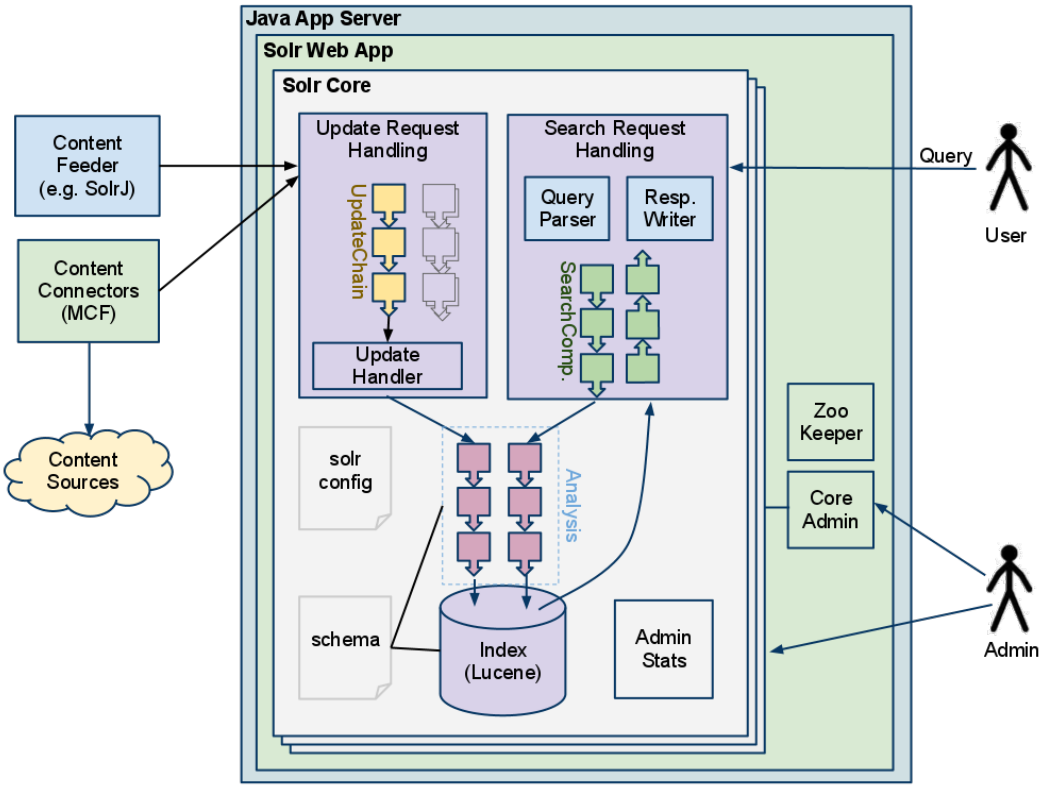


Figure 2.15: *Apache Solr: Architecture*

different way by Apache Solr. Each field is defined into the XML configuration files. The field type tells Apache Solr how to interpret the field and how it can be queried.

2.4.1.2 Configuration files

The most important Apache Solr configuration files reside inside the `$SOLR_HOME\conf` directory and are in the XML format:

- **schema.xml:** gives a general description of the type of the document that can be indexed via Apache Solr. It lists the field type definitions and all the fields that each document indexed could contain. Each field references a particular type. The schema must describe a primary key and a default search field. The primary key is necessary to uniquely identify each document indexed.
- **solrconfig.xml:** contain lots of parameters that can tweak Apache Solr functionalities. One of the most important element that can be defined in this XML file is the request handler. The request handler handles all the requests made to Apache Solr. Each command that regards indexing a new document is made via HTTP POST method. Quering Apache Solr is easy, each command

sent is a HTTP GET request. There could be a different handler for different purposes and each handler is registered against certain URI.

2.4.1.3 Schema

The first section of the schema is the definition of the field types: the data types. This section is between the `<types>` tags. The field type declares the type of the field such as boolean, numeric, date, and few kind of text. They are referenced by the field definition under the `<fields>` tag. An Example of the boolean field type is shown in the Fragment 2.2. Each field can be:

- **Indexed:** this data should be searchable or sortable.
- **Stored:** the field is eligible for inclusion in search results.
- **Multivalued:** if the field can contain more than one value.

Each field has a name that must be unique and a type that references one of the field types. Fields can be static or dynamic: if the name field is explicitely declared into the schema, the field is considered static. If the name of the field defined into the schema contains a wild card then is dynamic. During indexing time if a document contains a field that is not matched by an explicit field definition but does have a name matching with a dynamic field pattern it gets processed according to that definition .

Listing 2.2: *Fragment of the Schema.xml*

```
<fieldType name="boolean" class="solr.BoolField" sortMissingLast="true" omitNorms="true" />
```

2.4.1.4 Indexing data

The main communication between an application and Solr is over HTTP protocol however Apache Solr provides more channels to send and receive data. One example could be the remote streaming: Apache Solr will receive a URL that will be resolved instead of receving directly the data. Generally the URL represent a HTTP URL but it could be also a file system URI if the data is already present on the Apache Solr's machine. One interesting option is to use a `DataImportHandler`: data is fetched directly from a database. The data formats supported by Apache Solr are Solr-XML (a specific XML schema used to specify documents and their fields), Solr-binary (a binary representation of the structure), CSV (character separated value format) and rich documents like PDF,XLS, DOC, PPT.

Importing and deleting data from Apache Solr is not considered real-time. Apache Solr requires a commit after each major operation like importing or deleting documents from the index. Unlike a database, there are no distinct sessions between each client, and instead there is one global modification state. All uncommitted changes can be withdrawn by sending Apache Solr a `rollback` command. Apache Solr's index is internally based on a Lucene index. The index is composed of one or more segments. Modifications get committed to the last segment. Lucene will on occasions either start a new segment or merge old segments together in one. If Lucene has just one segment, it is in an optimized state, because each segment degrades query performance.

2.4.1.5 Searching data

Apache Solr provides two main ways to query the index: either with a simple HTTP software library or with an Apache Solr client API. In both cases, the user will send an HTTP GET request to the Apache Solr server instance providing an URL. The URL self describes all the attributes, the values and some optional parameters related to the search. The URL describes also the web application context where Apache Solr is installed on the Java Servlet engine and the request handler chosen during the search.

All the parameters related to the search are separated from the actual query. Each attribute and value that is searched is represented on the query string with the pair `Key=Value` [`K=V`].

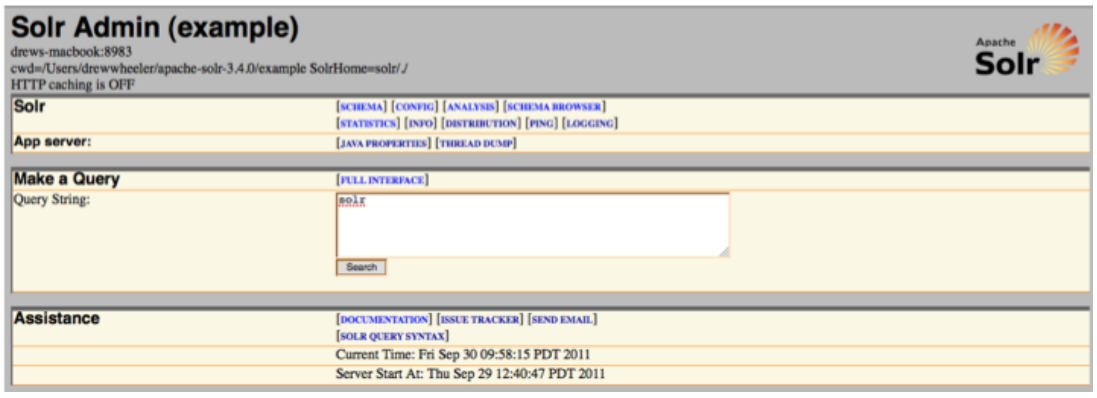


Figure 2.16: *Apache Solr admin GUI*

2.5 Monitoring

2.5.1 Nagios

Nagios is an open source tool specially developed to monitor host and service and designed to inform of network incidents before the clients, end-users or managers do [11]. It has been designed to run under the Linux operating system, but works also under most *NIX variants as well initially developed for servers and application monitoring, it is now widely used to monitor networks availability. Nagios works with a set of *plugins* to provide local and remote service status. The monitoring daemon runs intermittent checks on hosts and services specified using external *plugins* which return status information to Nagios. When incidents are detected, the daemon send notifications out to administrative contacts in a variety of different ways (email, instant message, SMS, etc.). Current status information, historical logs, and reports can all be accessed via a Web browser via the web GUI.

Figure 2.17 shows the Nagios architecture. The main components are :

- **Scheduler:** the server that checks the plugins and according to their results do some actions.
- **GUI:** the interface of Nagios. It allows the configuration of the scheduler and it displays the status of the plugins in web pages generated by GGI.
- **Plugins:** they can be configurable by the user. They check an external service and return a result to the Nagios server.

Nagios runs on a server, usually as a daemon (or service). Nagios uses a set of plugins and it runs them usually on the same server. Each plugin has a distinct function such as contacting hosts and servers on a local network or on the Internet. Nagios can also receive information from services and they can be checked through a web interface. Each time an error occours Nagios notifies the administrators through email or SMS. The notification system architecture is made of event handlers that can be configured to act each time something happens.

A soft alert is raised when a plugin returns a warning or an error. When a soft alert is raised many times (the number of times is configurable) a hard alert is raised and the Nagios server sends notifications: email, SMS etc.

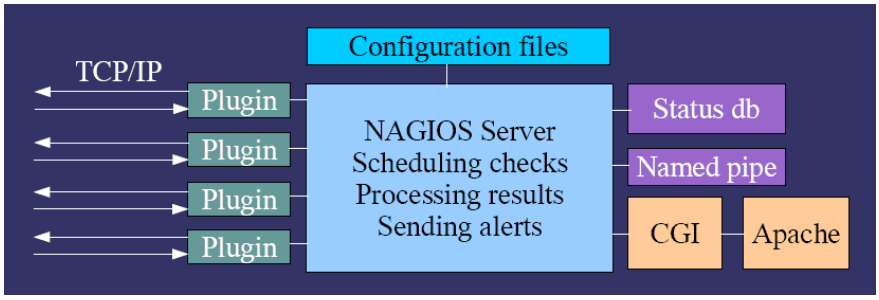


Figure 2.17: *Nagios: Architecture*

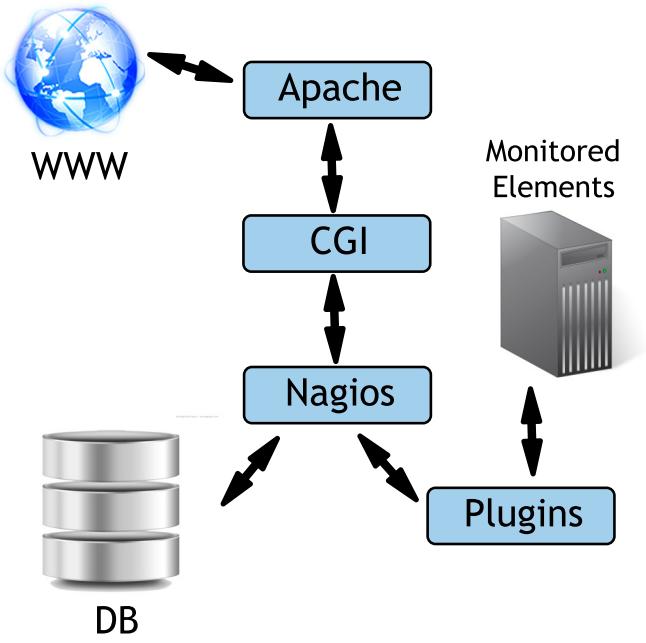


Figure 2.18: *Nagios: Internal Architecture*

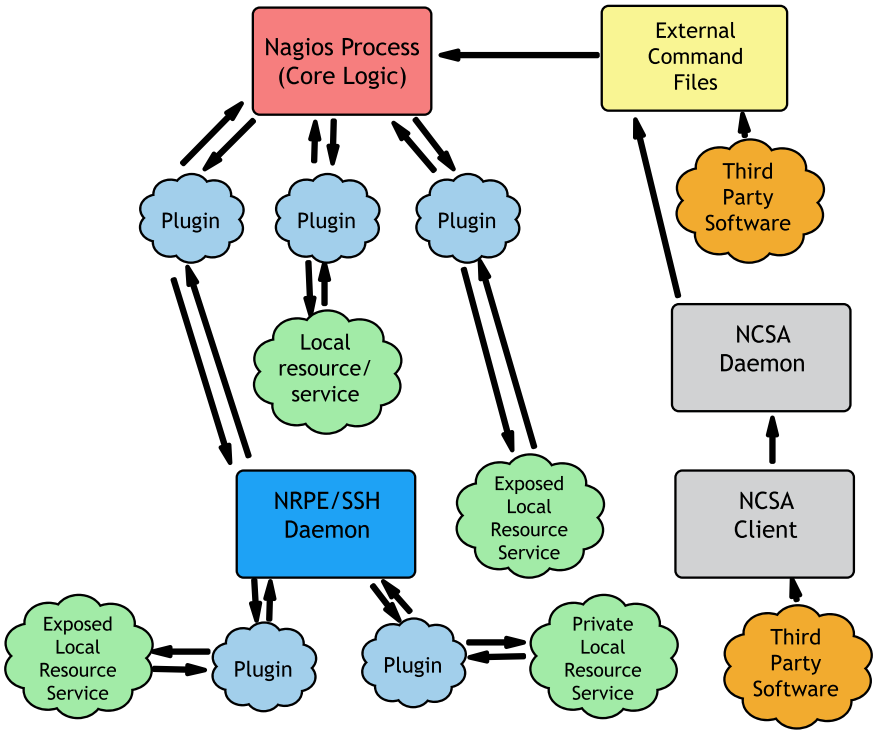


Figure 2.19: *Nagios: External Architecture*

Chapter 3

Design of the component

This chapter examines the design of the project. The purpose of this chapter is to give an high level description of the component starting with the requirements of the project and going on with a high-level description of the system without going into implementation details of single components.

3.1 Requirements

Different requirements of the application have been decided during the design phase of the project. Most of them are related to the environment in which the application will reside, but others are particular features requested from the company such as security, fast recovery and scalability.

3.1.1 Integration

The environment contains a number of existing applications and tools that the new application will use during the normal work flow. Most of these components are related to the current system architecture. These applications or actors are shown in Figure 3.1:

- **Data cycle:** it's the process that provides information to the environment. The data cycle can be seen as a daemon that grab information from the cloud and it makes it available in the form of raw data. It is one of the most important actors inside the company. All the data output are physically saved for major internal computations but it has also to be quickly reachable from other internal components such as bot servers and webservers. That is why all the data is also stored and indexed using search servers.

- **Search servers cluster:** these servers take the information provided by the data cycle, index them and makes them accessible and searchable through a API.

- **Database:** it contains all the saved search stored by the users.

- **Notification System:** this element has the duty to notify the users the newest information. The system is plugged with the email protocol, but it can have access also to the mobile protocol.

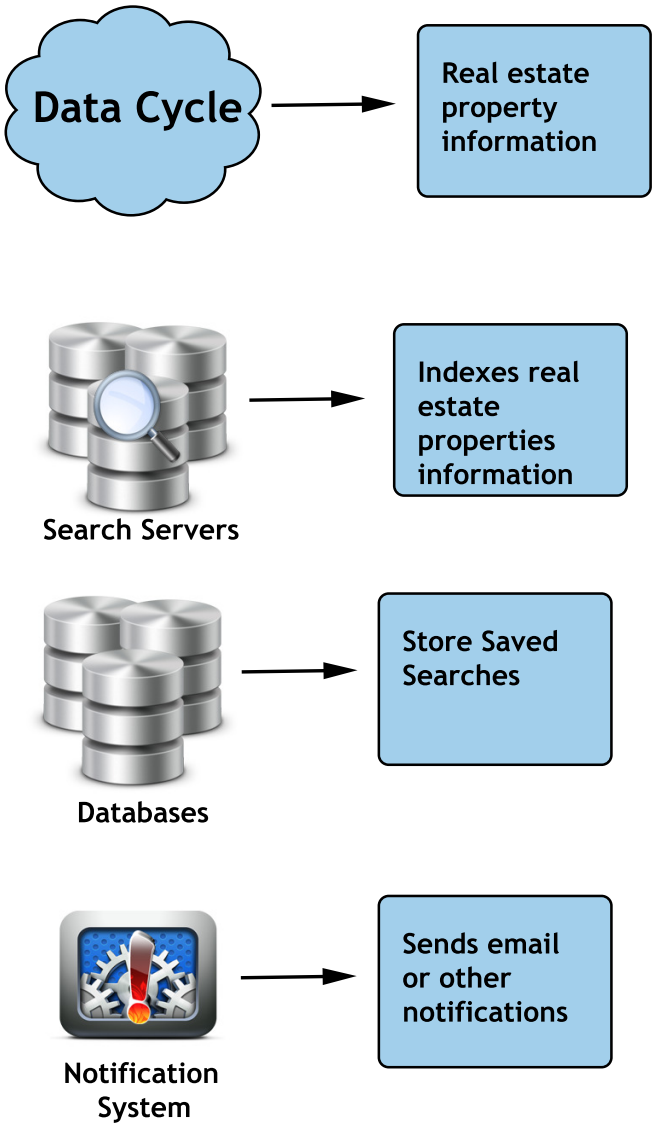


Figure 3.1: *Component actors*

The application has to communicate with the data cycle in order to acknowledge every time that a new information is available. The nature of this communication will be asynchronous and the application will not interfere with the normal work flow of the data cycle. The application could communicate with the search servers that contain the indexed data provided by the data cycle. The communication between these two actors should be as light as possible in order to avoid a bigger load that could slow the performance of the search servers. The application must be integrated in the system but it has to be easy to detach it from the environment in case of needs.

3.1.2 Resource Handling

Most of the actors resources of the system need to be used carefully in order to prevent starvation, deadlock or simply a overall worsening performance. In the current situation, the batch process execute every night each saved search against the search servers. This operation loads dramatically the search servers, raising the response time of an order of magnitude as said in the Chapter 1.1.2 and shown in an example graph in Figure 3.2 and Figure 3.3 .

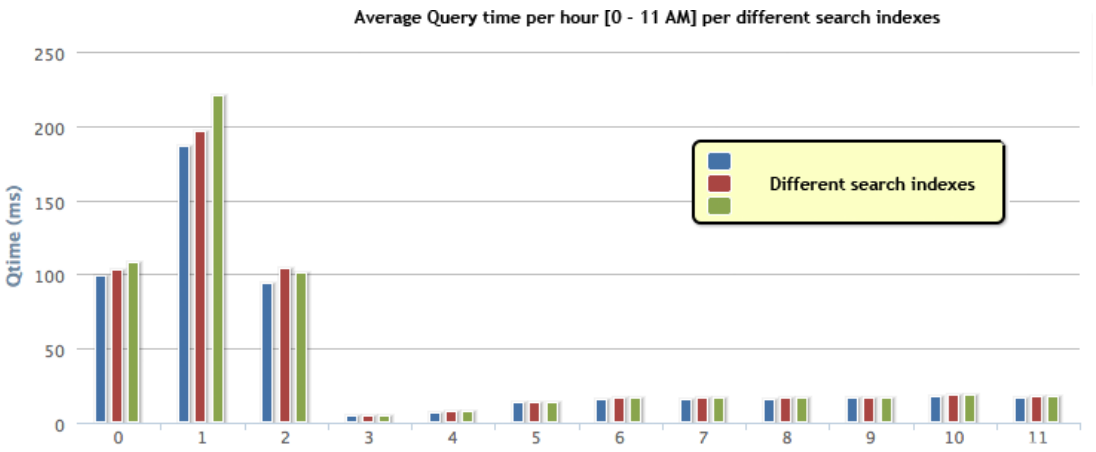


Figure 3.2: *Apache Solr search servers: average query time response during night time*

3.1.3 Scalability

Due to the rapid increase of the user base and of the projection of the growth of the interaction between the user and the web service provided by the company, the scalability of the application is a very important requirement. There are two different way to scale this application:

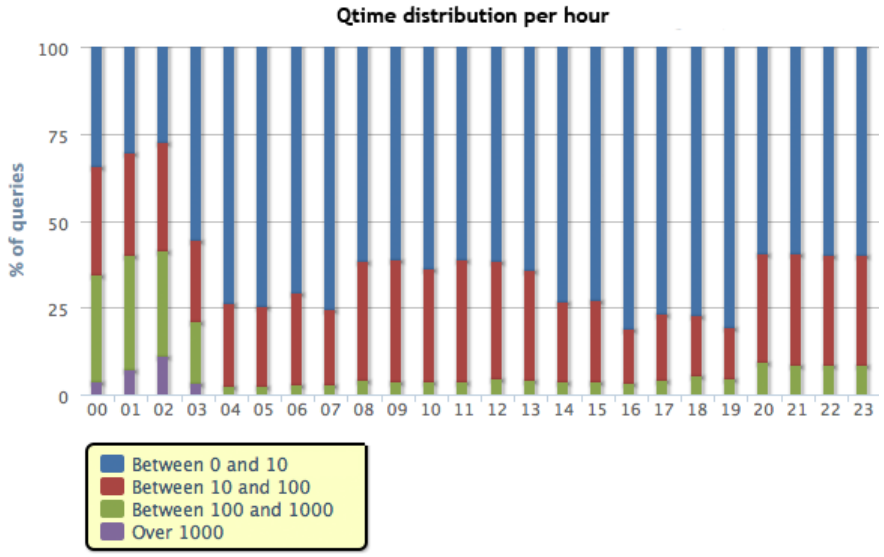


Figure 3.3: *Apache Solr search servers: query time response distribution during night time in percentage*

- **Horizontal:** the application provides better performance if the service is replicated and distributed on multiple machines. This requires that each replica of the service works with atomic operations while the resources are shared.
- **Vertical:** the application provides better performance if there are multiple instances of the service on the same machine.

3.1.4 Control Flow Application

The application is integrated with many actors in the environment and the flow of the service has to be monitored and controlled and it should be possible to detach the application from the system. The application can be plugged to monitor component that is already used in the system. In addition a web interface it is required to monitor the flow of the communication between the service and the other actors inside the environment.

3.1.5 Prospective search

The current system uses an open source technology to index and provide searchable information. One of the most important requirements of the project is to use the rapidity of this technology to provide the information needed to the real-time notification system. The new alert notification systems will use a reverse search method called *prospective search*. Since the current technology does not support

natively this feature, the purpose of this project is to investigate the possibility to implement it in the core or using an external custom plugin. The new functionality must not lower the performance of the overall system.

3.1.6 Open Source

The entire project uses tools and technologies that are released under the open source license. Since the current architecture of the company uses open source products one important is to maintain this line guide. All the new features that will be designed and implemented will be realeased under a open source license to encourage further developments. In particular, since the prospective search on the Apache Solr platform is a new feature and is not yet supported natively on this product, the ultimate goal is to prepare a generalized version of the implementation of the feature and commit it to the Apache Solr repository. This potentially represents one of the big innovative points of this thesis and of this project because it means contribute to a large scale open source project with a major feature.

3.2 Publish/Subscribe Overview

A handy way to give an high level description of the project is to refer to the hybrid *publish/subscribe* (P/S). As already said in the previous chapter, this model needs a *message broker* (MB) and queues to store messages and forward messages to the components. After considering various open source solutions on the market, it has been decided to use ActiveMQ. The main advantage of using this technology it is that it offers natively *Java Message Service* (JMS) as a *message oriented middleware* (MOM), and therefore provides a lot of services useful to the life cycle of the application.

Following the architectural model proposed by P/S, the complete system can be divided into two main phases. In the first phase, a user (**subscriber**) is interested in a class of information and decides to subscribe in order to be notified every time something new matches his interest. The second phase begins a flow of new information added to the system by the **publisher**. If the information matches the interest of one or more subscribers they have to be notified. The key of the system is the decoupling between subscriber and publisher. The publisher does not pick the message recipient (subscriber), instead a broker filters all the news and it handles the sending phase. Similarly, subscribers don't know who the publishers are nor have knowledge about the architecture of the system.

3.2.1 Subscriber

Each subscriber is a user who is surfing the website. After searching about real estate properties the user decides to subscribe to the service, saving his search using a handy pop-up as shown in Figure 3.4. Every search on the website is also reachable via URL. After a search is saved, the URL is stored in a MySQL table called *UserQuery* with additional information about the search such as a timestamp and the user unique identification. It is really important to remember that the MySQL table is already present in the current architecture and is used directly by the previous In order to successfully complete the subscriber phase, an automatic service takes the URL from the database table and after a transformation process, indexes against the prospective search index (PSI). Figure 3.5 shows the implementation scheme of the subscriber component.

Every saved search (from now on referred as URL) is processed from two main components:

- **URL Parser (URLP)**
- **Query documents generator (QDG)**

The URL Parser has the main purpose of understanding the nature of every URL and translate, where possible, any information contained in the search in a list of key-value pairs <K,V>. The QDG takes this list and creates a new document according to the Apache Solr document format. The document describes extensively the interest saved by the original subscriber. Subsequently, the QDG indexes the document against the prospective search index.

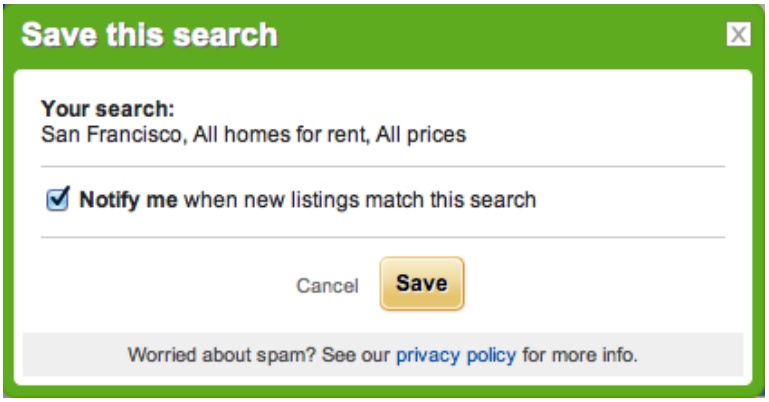


Figure 3.4: *Subscriber : pop-up window that allows to save search*

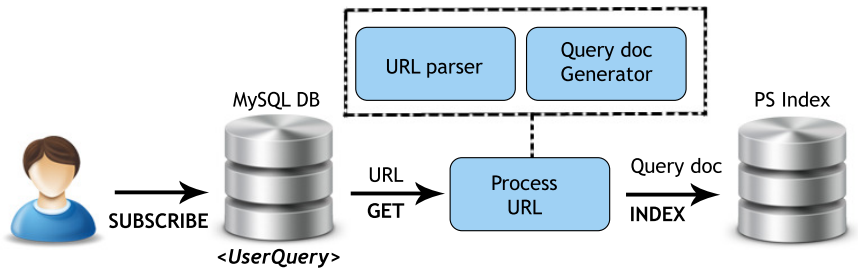


Figure 3.5: *Subscriber: architecture of the component*

3.2.2 Publisher

As mentioned earlier, the publisher is responsible for publishing new information within the system. As shown in Figure 3.6, the publisher is composed of the external service called *data cycle* which takes care of generating a new document each time there is news on a property. The event is triggered whenever a new property appears on the website, or there are major changes on the existing properties.

The document that is generated from the *data cycle* describes extensively all the information regarding the property. Each time the property is ready to be displayed on the website, the process also sends a new message containing the document on the queue *new documents*. Each message in the queue is extracted from the prospective search service. This service is responsible for querying the prospective search index asking what subscribers are interested in the document. To ensure that the lookup is efficient and correct, a special technique called **prospective search** is used by the prospective search service. The response of the request is processed by the service which is responsible for delivering a message on the *prospective search response* queue. The message contains the description of the original document that triggered the event and a list of all subscribers who are interested in the document.

3.2.3 Notifier

As shown in Figure 3.6 the notifier is responsible for extracting the message from the *prospective search response* queue and after processing the necessary information sends notifications to all subscribers who need it.

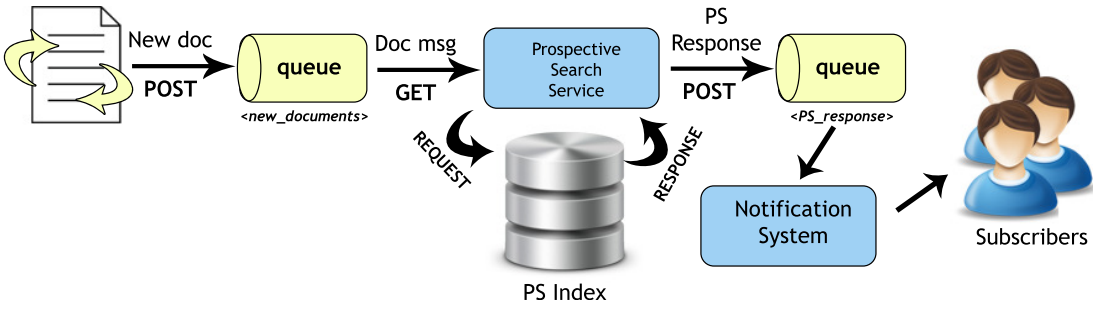


Figure 3.6: *Publisher and Notifier: architecture of the components*

3.3 Components Overview

3.3.1 Data cycle

The data cycle is the cloud that provides information about real estate properties. It is a affirmed component inside the company architecture. Practically the component can be seen as a deamon that keep releasing new information. The information can be divided in three groups:

- **New information:** it is a complete new document on a new real estate property that is not present in the company databases and indexes. This information will be inserted in the system.

- **Updates:** it is a update on a existing real estate property that is present in the company databases and indexes. The property document has to be tracked in the company databases and indexes and the new information must update the existing records.

- **Old information:** regards all the information that is already present in the company databases and indexes and it is not necessary to update or add it.

3.3.2 Database

The database is inherited by the previous architecture and it has the function of storing user saved search. The integration with the old system requires that the saved search are firstly stored on the database. The new architecture provides also a indexing system, so as second step the information is stored and indexed by the indexing system and the search servers.

3.3.3 Documents

The document is an important element of the system: it is the physical representation of the information given by the *data cycle* or the physical representation of the saved search. In both cases, the document is a simple format that describes all the elements contained in the information. The usual format is a list of property or key and value `<K,V>`. Each document has an unique identification (ID).

3.3.3.1 Real estate property doc

This document is created and released by the *data cycle* and describes a real estate property. Each document has an unique identification called property ID. This ID is used by the system to understand if the property is already present in the databases and the indexes.

3.3.3.2 Query doc

This docuement is the representation of a saved search and is created by the URL processor. Since each saved search is a list of unordered facts to which the user is interested, the URL processor translate all the facts in a list of pairs key value `<K,V>`. Each document has a unique identification called query document ID and each key is unique in the document. That means that if a saved search contains multiple elements pointing to the same attribute, the query document will contain a single key and multiple values pointing to the same key.

3.3.4 URL processor

The URL processor takes a saved search from the database and transforms it into a query document. The component is made of two subcomponents called URL parser and Query Document Generator. The URL processor is connected to the database and receives the saved search. The search is stored on the database under the form of URL since each URL contains all the search parameters that identifies a particular search. At this point the component processes the URL and extracts the information needed to create a query document. The URL processor processes also all the information related to the search that would be useful later to connect the URL to the User such as the User identification and his email.

3.3.5 Prospective Search Index

The prospective search index is responsable to store and index all the query documents. Each query documents represents a single saved search stored also in the databases. The purpose of using an index instead of a normal database is that is more responsive and the performance are more relevant in this phase of the system because the overall component is search intensive. The index should receive and store query documents from the URL processor and communicates with the prospective search service each time a prospective search is needed.

3.3.6 Prospective Search Service

It represents the plumbing between the index, the notification system and the data cycle. Each time a new property document comes out from the data cycle the service must query the index and discovers which query documents match the property document. Basically the service implements the prospective search using the index as support. After collecting the results from the index the service communicates them to the notification system in order to send notifications to the users.

3.3.7 Communication between components

The communication between the components has to be decoupled and avoid serializability. The solution is to use a queue system that interconnect all the components. In particular the prospective search service has to be connected with the data cycle and also with the notification system. Due the position of the two queues they are referred as input queue and output queue. The last queue will be a signal queue to signal any error occurred .

3.3.8 Notifier service

The notifier system has to take a prospective search service response and create the notification. The system supports email notification and the response of the service contains all the information that is necessary such as the user information and email address, the saved search triggered by the property document and the original property document. The service which is a daemon uses a defined model to prepare the messages and finally sends right away the email to the user.

Chapter 4

Implementation

This chapter examines the implementation of the project. In particular, attention is placed on how each component of the system is built and how it interacts with the rest of the system.

4.1 Prospective Search index

One of the basic requirement of the project is to implement a real-time notification system using the Apache Solr technology and the prospective search (PS). At this time, Apache Solr does not support this feature: in order to fulfill the requirement it's possible to modify the basic configuration of the product and, where necessary, extends the functionality using ad-hoc plugins.

4.1.1 Schema

In order to support the particular format of the query document which is stored into the prospective search index, some modifications have to be done to the Apache Solr prospective search index *schema.xml*. First, each query document structure is dynamic because it depends on the user search: it's not possible to describe every possible field in the schema. In order to solve this problem, we use dynamic fields where each definition of the field is based on the type, not on the content. The main classes of dynamic fields are:

- **String fields _s:** each field that contains a string value. These fields are stored normally in the index.
- **Numeric fields _i ,_f :** each field contains an integer or a float value. These fields are stored as the special trie field. This kind of field allows better

performance while using special search tecniques like faceting searches or range searches.

- **Boolean fields _b:** each field that contains a boolean value.

.

Listing 4.1: *Fragment of the configuration file schema.xml*

```
...
<dynamicField name="*_i" type="tint" indexed="true"
stored="true" multiValued="true"/>
<dynamicField name="*_f" type="tfloat" indexed="true"
stored="true" multiValued="true"/>
<dynamicField name="*_s" type="string" indexed="true"
stored="true" multiValued="true"/>
...
```

Due to the unpredictability of the query documents each field is stored as multivalued. Second, every field has to be searchable, but just the fields related to the subscriber information and the subscription has to be stored into the index. In order to save space on the index, only the fields related to the subscriber are stored into the index.

4.1.2 Data Import Handler

Each time a subscriber saves a search on the website, the URL of the search is stored into a MySQL table called *UserQuery* with other information such as a timestamp. As already mentioned, in order to complete the subscription for that particular search, a service has to take the URL from the database, process it and then store it against the prospective search index.

Apache Solr provides a handy tool that can import data from an external source such as a database. This tool is called **data import handler** (from now on called DIH), and provides an automatic way to import data into the Apache Solr index. This feature is really important but it has to be modified to be used with the prospective search index. The solution is to design and implement a custom data import handler that can be adapted to work with the prospective search index. The components of the tool (Figure 4.1) are :

- **URL parser:** extracts the URL from the database table. Its duty is to comprehend the nature of the URL and transpose all the information contained in a key-value <K,V> list.

- **Query Document Generator :** takes all the couples <K,V> related to a URL that are passed from the URL parser. Its duty is to create a query document and to index the document against the prospective search index.

An extra configuration is needed to tell where and how take all the data from the source. The XML configuration file called *data-config.xml* contains:

- **Data source :** it specifies the URI of the source and the driver that has to handle the connection. In the specific case of this project, the driver used is a JDBC connector. Inside the same tag descriptor there are also the database name and the credentials.
- **Query:** the SQL query statement used during the full-import feature offered by the DIH.
- **DeltaImportQuery and deltaQuery:** the SQL query statements used in the delta-import feature offered by the DIH.
- **DeletedPkQuery:** the SQL query statement used during the delta-import feature offered by the DIH to syncronize deleted elements in the source.

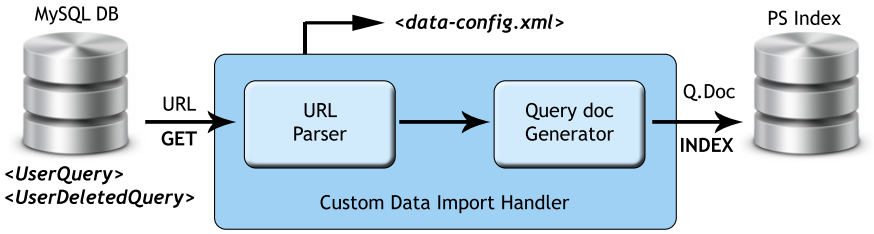


Figure 4.1: *Custom Data Import Handler: architecture of the component*

4.1.2.1 DIH features

One of the major advantages of the DIH is its complete integration within Apache Solr. Each import operation can be considered atomic. This means that each error that results in an abort does not bring the index to an inconsistent state. A rollback function is automatically called after a system abort since every import is an indipendent transaction. Also, during the import phase, the prospective search index can be used to search using the last consistent data. After and only in the case of completed import, the new data is immediatly available.

The import process can be triggered with a simple HTTP request to a URL provided by the prospective search index:

.

Listing 4.2: *Custom Data Import Handler: URL used to trigger a command*

```
http://prospective searchHostname:prospective searchPort/dataimport?command=NameOfTheCommand
```

The list of the commands available for the DIH are:

- **Full-import :** it triggers the full-import.
- **Delta-import:** it triggers the delta-import.
- **Status:** it gives information about the current running import process.
- **Abort:** it aborts any running import process and triggers the rollback functionality.

The status command provides a JSON or a XML response each time is used. An example of the response of the DIH is shown in Figure 4.2. A cron job is used to trigger the real-time import of the DIH.


```
- <response>
  - <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">0</int>
    </lst>
  - <lst name="initArgs">
    - <lst name="defaults">
        <str name="config">data-config.xml</str>
      </lst>
    </lst>
    <str name="command">status</str>
    <str name="status">idle</str>
    <str name="importResponse"/>
  - <lst name="statusMessages">
      <str name="Total Requests made to DataSource">2</str>
      <str name="Total Rows Fetched">0</str>
      <str name="Total Documents Skipped">0</str>
      <str name="Delta Dump started">2011-12-04 22:20:01</str>
      <str name="Identifying Delta">2011-12-04 22:20:01</str>
      <str name="Deltas Obtained">2011-12-04 22:20:02</str>
      <str name="Building documents">2011-12-04 22:20:02</str>
      <str name="Total Changed Documents">0</str>
      <str name="Total Documents Processed">0</str>
      <str name="Time taken ">0:0:0.650</str>
    </lst>
  - <str name="WARNING">
      This response format is experimental. It is likely to change in the future.
    </str>
</response>
```


Figure 4.2: *Custom Data Import Handler: XML status response*

4.1.2.2 Full import

The *UserQuery* MySQL table contains a lot of data. The full-import feature imports all the data contained in the source in a single transaction. Given the dimension of the data stored in the source, the full-import can not guarantee a continous update of the index. This feature can be used the first time to preload the data on the prospective search index when it is empty and does not contain a single query document. To ensure a real-time update of the index the delta-import feature is used.

4.1.2.3 Delta import

The delta-import solves the problem of importing a big amount of data from the data source while allowing real-time update of the index. With this feature, the DIH imports just the data that is not already present in the data source. Every time that it is triggered, it first checks information about the last successful delta-import. Each time that a delta-import succesfully completes an import the DIH saves additional information about the operation. The most important are the timestamp of the beginning of the import and the id of the last saved document on the index. With this data, the DIH can always check what is the delta between the data source and the index and start importing from the last successfull checkpoint.

Al already mentioned, this feature has to be configured properly in order to work. The *data-config.xml* file contains two sections related to the delta-import feature called *deltaImportQuery* and *deltaQuery*. *deltaQuery* is the SQL query statement that describes how to return all the unique ids of the new rows on the data source table that is *UserQuery*. *deltaImportQuery* is the SQL query statement that describes how to get row by row each result given by the *deltaQuery*.

The delta-import is considered finished when all the rows are examined. At that point, the delta import is concluded and the DIH saves the timestamp of the successful import and the id of the last query document imported. If a small error on a query document occurs the DIH will notify that in the status response and it will skip the document, but if a major error occurs an abort will be sent and a rollback will be triggered. In this case no data about the current import will be stored and indexed by the DIH. The next import will begin from the last successfull import.

4.1.2.4 Update and delete documents

When a subscriber changes the status of one of his subscription, the modifications are stored in another table called *UserDeletedQuery*. The subscriber can delete his

interest or make his entry inactive. In order to maintain the prospective search index in a consistent state this changes must be carried from the database table to the index. The *deletedPkQuery* has this purpose, the SQL statement describes how to extract all the inactive and deleted URLs and how to delete the documents from the prospective search index that are not useful anymore. Below is shown a fragment of the data-config.xml that contains full import, data import and deleted queries. .

Listing 4.3: *Custom Data Import Handler: fragment of the data-config.xml configuration file*

```
<dataConfig>
<dataSource type="JdbcDataSource" driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://Hostname/DatabaseName" convertType="true" batchSize="-1"/>
<document name="prospective search Doc">
<entity name="UserQuery" onError="skip" pk="id" query="full-import subscriptions"
deletedPkQuery="delete removed and inactive subscriptions"
deltaImportQuery="select every single subscriptions matched by deltaQuery"
deltaQuery="select all subscriptions with subscription date greater than last index time">
</entity>
</document>
</dataConfig>
```

4.1.3 Cron Job

The cron job triggers the DIH on a certain scheduled time. In order to maintain the prospective search index syncronized with the *UserQuery* table, the cron job triggers the delta import of the DIH every minute. .

Listing 4.4: *Custom Data Import Handler: fragment of the cron job configuration*

```
*/1 * * * * curl -s 'http://prospective searchHostname:prospective searchPort/dataimport?command=delta-import'
```

4.2 URL parser

This part of the component takes as input a URL and returns as output a JSON file. The JSON file describes extensively all the information on a single search: it contains a list of all the fields and values related to the search. However, some searches are complex and require further steps before being transformed into a query document and indexed by the prospective search index.

4.3 Query document generator

The input of the query document generator (from now on QDG) is the JSON file passed by the URL parser. The purpose of the QDG is to generate a query document.

The SolrJ client library is used to generate a valid Solr document and index it against the prospective search index.

Some of the URLs parsed regards simple search: the JSON returned by the URL parser tool in this cases already contain all the information to create the document. In other cases, the URL rappresents a special search (from now identified also as query), like a range query or a spatial query. The QDG handles this searches and stores the appropriate document inside the prospective search index. Figure 4.3 shows that is possible to identify three main classes of searches that require a special treatment in order to be correctly stored inside the prospective search index.

The simplest class of search is called *boolean queries*: it contains all the boolean conjunctive queries. The second class is called *range queries*: it contains all queries that consists of search on numerical fields, or dates and implies ranges. The last class is called *spatial queries*: it contains all the queries regarding geolocation. There may be a *hybrid search*: if it is possibile to identify multiple classes of queries inside a single search. In this case multiple techniques must be combined in order to process correctly the search.

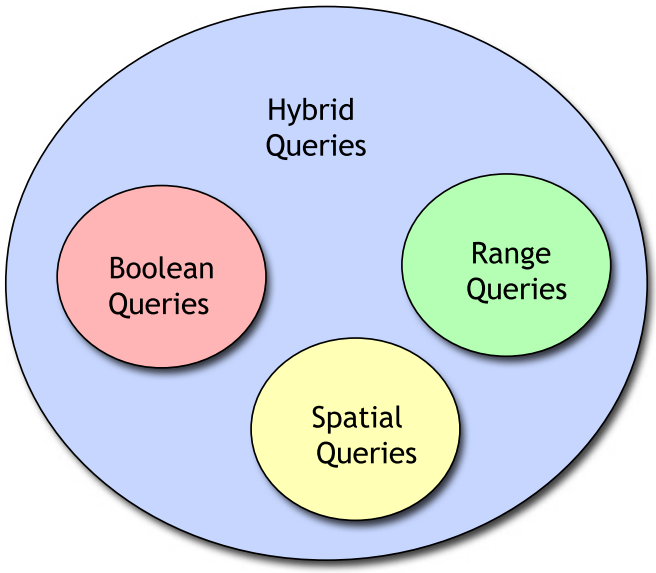


Figure 4.3: *Query Document Generator: main classes of searches*

4.3.1 Boolean conjunctive queries

A boolean conjunctive query is a conjunctive query without distinguished predicates, i.e., a query in the form $K_1(V_1) \wedge \ldots \wedge K_n(V_n)$ where each K_i is the key (name of the field) and each V_i is the value related to the key. Due to the simple structure of

this class of queries, the QDG simplifies the chain of the fields and values in order to get the correct list of elements. It is possible to identify an expression as a pair key-value $< K, V >$. From specifications, all the saved search contains affermative expressions. The only exception of this rule is when a value related to a key is negated ($< K, \neg V >$): in this case the expression is the same as $< K, V_i >$ where V_i is every value other than V. The expressions are connected through the logic connector AND. Figure 4.4 shows an example of how the QDG handles this class of queries.

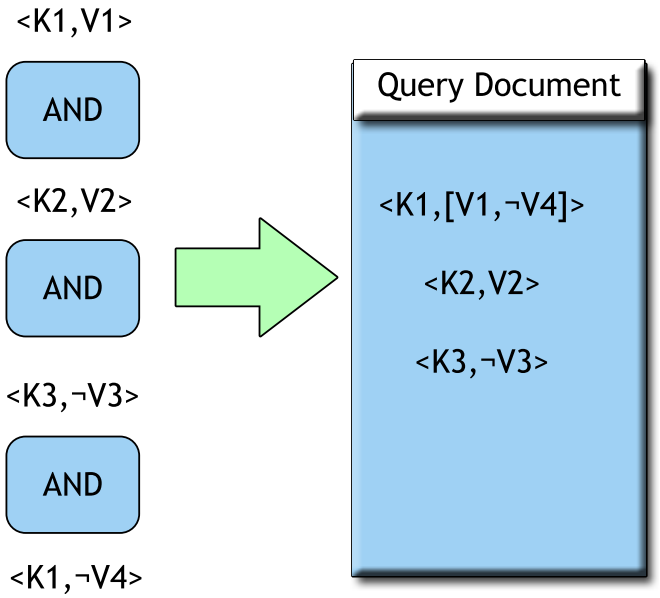


Figure 4.4: *Query Document Generator: boolean queries*

4.3.2 Range queries

A range query is a query that is designed to retrieve all records where some value is between an upper and lower boundary. Often the value considered is a numeric field such as integer or float, but it can also be a date. Figure 4.5 shows how the QDG handles this class of queries. First, the QDG checks if the query defines both the lower and the higher bounday: it is common that a subscriber's search describe an interval where just one between the lower and the upper bound are defined. Each time the URL parser gets a search with a single bound, it writes a star (*) in the JSON response to identify that the current field could assume each value. As shown in the Algorithm 1 each field is divided in two new fields that represent the boundaries of the orginal. If one of the lower and the higher boundary is not present then is not reported in the query document.

Algorithm 1 Query Document Generator: algorithm for processing range queries (K,<min,max>)

1: $min_k \leftarrow null$
2: $max_k \leftarrow null$
3: **if** $((boundary(min, K) \neq NIL)$ and $(boundary(min, K) \neq *))$ **then**
$min_k \leftarrow boundary(min, K)$
4: **end if**
5: **if** $((boundary(max, K) \neq NIL)$ and $(boundary(max, K) \neq *))$ **then**
$max_k \leftarrow boundary(max, K)$
6: **end if**
7: return min_k, max_k

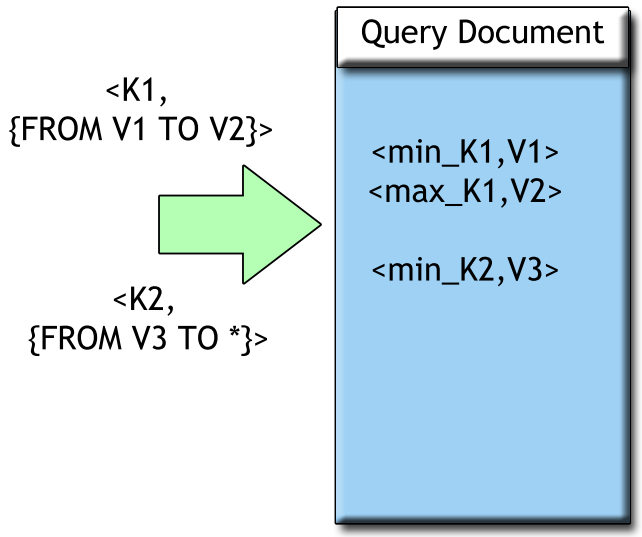


Figure 4.5: *Query Document Generator: range queries*

4.3.3 Spatial queries

A spatial query is a query that contains a geolocation. The QDG takes each field with a location and applies the algorithm shown by Algorithm 2. Depending on the number of coordinates provided by the subscriber in the search, the QDG will save the informations as a:

- **Point :** the coordinate regards just a point: in this case the fields related to the coordinate are longitude and latitude.
- **Bounding box and Polygon:** the coordinates are multiple, the minimum bounding box will be created using all the points. The 2D bounding box will be stored in the query document and used in a preliminary query information

retrieval. Since the bounding box is not precise with irregular geometric figure, a polygon is also stored inside the document in order to be used to filter the preliminary search.

Figure 4.6 shows how the QDG handle this class of queries. 4.6

Algorithm 2 Query Document Generator: algorithm for processing spatial queries (<coordinates>)

1: **if** $length(coordinates) = 1$ **then** $store\_point(coordinates)$
2: **end if**
3: **if** $length(coordinates) \geq 2$ **then**
$store\_bbox(coordinates)$
$store\_polygon(coordinates)$
4: **end if**

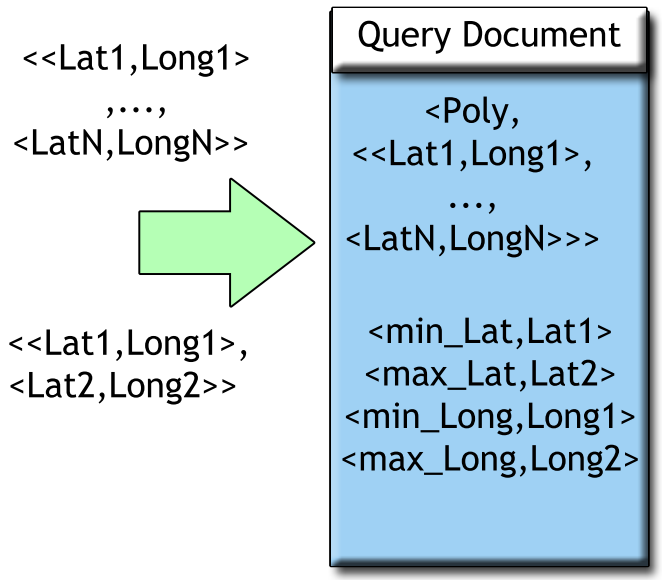


Figure 4.6: *Query Document Generator: spatial queries*

4.4 Queue System

After considering various products on the market, ActiveMQ has been chosen as the message broker. The message network has to be the plumbing between the *data cycle* and the *prospective search service*. It has also to furnish the connection between the *prospective search service* and the *notification system*. Figure 4.7 shows what queues are used in the system. The JMS (Java Message Service) API is used to access to the

queues and to the messages inside them. This additional level of abstraction is really useful because it totally decouples the queue system logic from the implementation. In the future, the broker technology could be changed without design again the entire system.

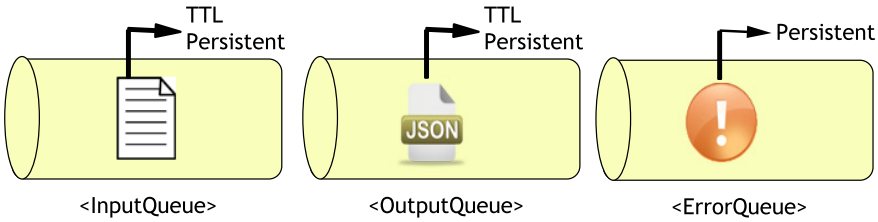


Figure 4.7: *Queue System: input, output, error queues*

4.4.1 Input queue

Each message contained in the input queue comes from the *data cycle* and the payload of the message contains a Apache Solr document. The document describes a new real estate property or an old one with few changes or new information. The queue itself and the messages within have some properties:

- **Persistency**: if the broker is not reachable due to a system crash or a network problem, each message has to remain in the queue. Ideally, each time the system is restored the queue should contain the same messages that contained before the crash. This feature is automatically handled by the message broker storing each message prior in memory and then physically on disk. Each time a message has to be forwarded, the message broker checks if the message is present in memory and if not it will load it in cache and send it. After the send phase, the message is deleted from memory and disk.

- **TTL**: if a message is not consumed within a certain time limit, it can be considered obsolete and not useful. This is related to the real-time nature of the system where each message it should be processed and forwarded immediately. If a message stays for a long time in the queue means that the overall system suffers of some kind of problem. If a message became obsolete it should automatically discarded by the broker. The TTL (Time To Live) has the purpose to give to each message a time limit of life , after that the message broker can get rid of the message.

4.4.2 Output queue

Each message contained in the output queue comes directly from the *prospective search service* and the payload of the message contains raw JSON data. Each message is compressed and optimized in order to limit the dimension of the text payload. The TTL and the persistency properties are applied on the messages.

4.4.3 Error queue

This queue is utilized as a signal queue by the entire system. If some error compromises the ability of the system to fulfill his purpose, a message is sent to this queue. The message payload will contain a reference error and the original document that was contained the input queue. This helps the system to recover: after that the error is is resolved, the original document can be reprocessed. No TTL policies are applied on the messages on this queue because each error message must be persistent until is read and treated.

4.5 Prospective Search service

It's the service that takes a message from the input queue and using the *prospective search* technique interrogates the prospective search index. After getting a response from the prospective search index, it creates a prospective search response in the JSON format. The response is attached as payload on a message and it is sent on the output queue.

4.5.1 Prospective Search on Solr

Apache Solr does not support the *prospective search* or any kind of reverse search. Figure 4.8 shows the normal functioning of Solr which is a retrospective search: before, a set of documents are stored in the index, then a search request containing a query is sent to the index . The result is a list of documents that match that query. Figure 4.9 shows what we want to accomplish: before, a set of queries are stored in the index, then a search request containing a document is sent to the index. The result is a list of queries that match the document. Figure 4.10 shows how we can implement the search with Apache Solr even if it does not support the reverse search: before, a set of query documents are stored in the index, then a search request containing a prospective search query is sent to the index. The result is alist of query documente that match the prospective search query. The prospective search query

is created following all the information inside the original document. Each query document describe a saved search.

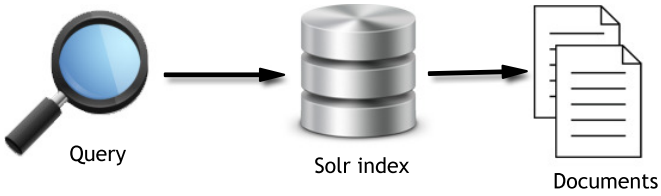


Figure 4.8: *Retrospective search*

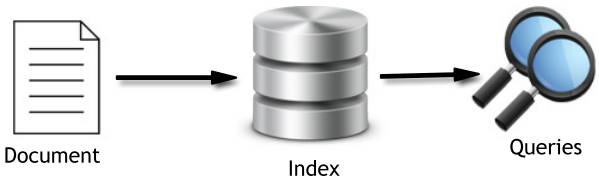


Figure 4.9: *Prospective search*

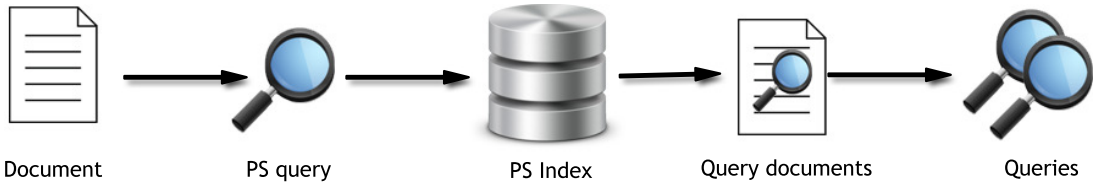


Figure 4.10: *Prospective search on Solr*

4.5.2 Request

The message that is dequeued from the input queue is used to create a prospective search request. The service extracts the document contained in the message (in the Apache Solr document format) and prepares a prospective search query that has to be executed on the prospective search index. The response of the prospective search index must return all the query documents that match the property document out of the set of registered query documents. In order to do that, the query sent from the prospective search service to the prospective search index must contain all the informations about the original property document. Since Apache Solr doesn't support this reverse search, the query built from the prospective search service is a special query that keeps in consideration how the saved searches are stored in the index.

4.5.2.1 Prospective Search query

As shown in Figure 4.11, the matching problem between the real estate property document (from now on called also PD) and the query documents that represents the saved searches (from now on called also QD) could be considered as a problem on sets. If we compare the property document with various query documents we can notice that each saved search that matches the document contains a subset of information of the document.

The prospective search query is made of expressions, all alternated with boolean conjunctions (AND). There is an expression for each element within the property document. Each element within the PD can be described as a key-value pair <K,V>, where in some cases the value is a list of items. Each expression is defined as the boolean disjunction (OR) between:

- **Key-value pair <K,V>**
- **Negation of the Key**

The combination synthesize two possible cases: the first one catches the case when the <K,V> in the property document is also present in the query document. It throws every situation when the query document has the same key K but different value V. The second one catches the case when the query document does not contain the key K : we must consider every correct subset of informations of the property document.

4.5.3 Post process

Once that the prospective search index ends the matching phase, it responds to the prospective search service with the list of query documents that match the property document. The prospective search service verifies if each document contain a special search class such as the spatial queries. If so, another phase is necessary: the prospective search service must check if that query documents match the original document based on the geographical information contained. Each false positive will be discarded.

4.5.4 Response

After the post process, the prospective search service has the exact list of the query documents that match the original property document. Each QD is identified by an id and a query hash. The prospective search service can now build a JSON response

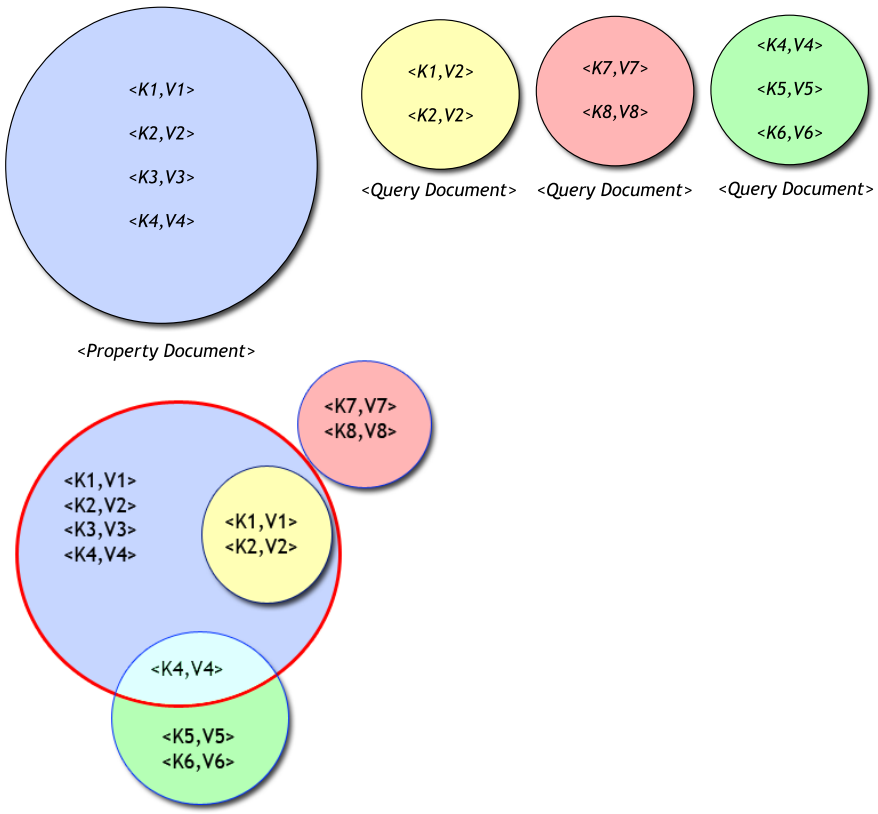


Figure 4.11: *Prospective search query: matching between property document and query documents.*

containing all the list of the query hashes and ids that represents all the interests of the subscribers. The JSON is then wrapped in a message and then sent on the output queue.

4.6 Notification System

A batch process pulls out each message from the output queue and it handles the sending phase of the notifications. The notification system must be adapted, however, in the case of real-time notifications:

- **De-duplication phase:** a subscriber must receive a single notification per updates even if the new property matches multiple interests.
- **Time frame window:** a subscriber should receive a single notification containing the last updates and not multiple notifications in succession. In order to solve this problem the batch process will collect messages within the time frame window and then it will send out the notifications a single time .

4.6.1 Email notification

Trulia.com has already a email notification system that sends daily and weekly updates to users. The simple format of message contained in the output queue, allows the email notification system to extract the data from the JSON and send an email to each subscriber referred in the JSON file. The layout and the information contained in the email is not a case study of this thesis.

4.7 Monitoring

Each component of the system is vital to the correct behaviour of the notification system. The monitoring tool called Nagios, provides an easy way to continuosly check the health of each component. In order to do so each element has to provide an accessible interface reachable from Nagios such as a web service. The tool will notify the administrator of the system if it encounter any kind of inaspected situation via email. The three main actors of the system that has to be monitored are :

- **Queue System**: ActiveMQ provides an easy integration with the Nagios monitoring tool. The focus of the monitoring must be placed on the persistance of the message broker service and on the single queues. The ActiveMQ message broker daemon has to be reachable at any time therefore the Nagios monitoring tool sends an HTTP request to the broker each minute and for an HTTP response with the correct status: `200 ok`. The *error queue* has to be empty all the time overwise means that a component in the system provoked an error. Nagios checks each 10 minutes the status of the *error queue* and if the queue contains messages it will consider the situation with three levels of warning priority:
  - **low**: if the message count is more than 1 and below 3
  - **medium**: if the message count is more than 3 and below 5
  - **high**: if the message count is more than 5

  The level of priority is notified together with some information about the messages. The input queue and the output queue contain a random number of messages so it is not possible to provide a strict rule about the count of the elements inside these queues. The only security monitoring that can be done on these two queues is to check every 10 minutes if the message counts go beyond a certain security threshold that has been decided to be of 10000 messages per each queue.

- **Prospective Search Service**: Nagios checks the status of the service sending a HTTP request to the URI exposed from the service. The service must be reachable every time so each minute Nagios sends a HTTP request to check the status of the service. The correct HTTP response is: `200 ok`. Other responses of a timeout related to the request are considered errors.

- **Prospective Search Index**: Nagios checks few functionality of the index, mostly testing the status and the performance of the index:

  - **Status**: Nagios sends a HTTP request to the prospective search index each minute and it checks if the index is reachable and responsive.
  - **Import**: is also important to maintain the index syncronize with the *UserQuery* MySQL table. Each 15 minutes Nagios checks the timestamp of the last succesful import notifying the Administrator if is too old.
  - **Response time**: Each 5 minutes Nagios checks the performance of the index.

Conclusioni e sviluppi futuri

Lo sviluppo di questa tesi ha presentato un lavoro di progettazione e di implementazione di un sistema di notifica real time che offre la possibilità di inviare un flusso di informazioni tramite email ad ogni utente registrato al servizio presso il sito web **Trulia.com**.

Il progetto realizzato ha avuto l'ambizione di introdurre una nuova feature all'interno del corrente sistema di notifica dell'azienda. Un'ambiente così complesso e frenetico come è quello del mercato immobiliare americano necessita di una continua interconnessione tra gli attori del sistema, ovvero tra gli utenti finali e gli agenti o brokers.

L'applicazione si occupa in tutta autonomia di recepire nuove informazioni su proprietà immobiliari e dopo aver individuato quali tra queste interessino all'utente invia ad esso una notifica attraverso email. Gli utenti registrano i propri interessi direttamente sul sito web, dopo aver utilizzato la normale ricerca fornita dal servizio. Essi vengono quindi salvati e indicizzati per una veloce consultazione da parte dell'applicazione sfruttando la piattaforma di ricerca Apache Solr. Ogni volta che l'applicazione recepisce una nuova informazione su una proprietà immobiliare, opera un'operazione di filtraggio logico su tutti gli interessi precedentemente indicizzati ed intuisce quali tra questi corrispondono alla nuova informazione. Per svolgere quest'ultima funzione durante lo sviluppo dell'applicazione è stata progettata e implementata la tecnica definita *prospective search* o *reverse search* sulla piattaforma Apache Solr che ne era sprovvista.

Per garantire disaccoppiamento tra tutti i componenti dell'applicazione è stata utilizzata una versione ibrida del pattern messaggistico *publish/subscribe* sfruttando JMS API come strato architetturale addizionale sopra all'implementazione svolta tramite l'utilizzo del MOM Apache ActiveMQ.

Durante lo sviluppo del progetto una grande importanza è stata data al tentativo di integrare diverse tecnologie per poter garantire un grado di innovazione pur mantenendo punti fondamentali come la stabilità e la sicurezza nel mondo enterprise. L'utilizzo di Spring Framework e dell'application container Jetty ha portato un grande

vantaggio. Entrambi hanno standardizzato la modalità di sviluppo dell'applicazione garantendo un possibile futuro ampliamento delle caratteristiche o un adattamento della stessa per altri scopi.

Il lavoro svolto comprende anche un'interfaccia accessibile tramite browser che garantisce la possibilità di controllare l'applicazione in tutto il ciclo di vita e di impostare una serie di parametri di configurazione che permettono di variare il comportamento del sistema. Il controllo del ciclo di vita dell'applicazione è svolto attraverso l'utilizzo di strumenti automatizzati come Nagios e l'utilizzo di logging mirato con Apache Lo4J. In ogni momento del funzionamento del componente è possibile comprendere il contenuto delle informazioni processate.

Attualmente il progetto è in fase di testing avanzato; una volta terminato questo periodo sarà a disposizione dell'azienda per poter essere rilasciato in produzione e fruibile dagli utenti.

Nonostante il progetto abbia sfruttato e collaborato con elementi già esistenti nel panorama architetturale dell'azienda, come il processo che si occupa di inviare le notifiche via email agli utenti, ha sviluppato in parallelo un architettura alternativa che è utilizzata nel presente ma che può offrire interessanti opzioni di sviluppo future. Ne è un perfetto esempio l'utilizzo di mobile device come piattaforma di arrivo di notifiche in tempo reale. L'azienda ha già lanciato sul mercato nel corso dell' ultimo anno applicazioni mobile per Iphone, Ipad, Android, Kindle Fire e possiede quindi già i mezzi per poter comunicare in tutta facilità con gli utenti. Grazie all'architettura disaccoppiata del componente realizzato e all'utilizzo di un sistema di messaggistica a code è facile pensare alla possibilità di espandere questo servizio anche nel mondo dei mobile devices permettendo un'integrazione ancora superiore. Ulteriori sviluppi futuri riguardano la possibile sostituzione del già collaudato servizio di notifica giornaliero e settimanale con il componente che è stato analizzato in questa tesi. L'attuale servizio che svolge questo compito è un batch process che semplicemente giornalmente o settimanalmente esegue tutte le ricerche salvate dagli utenti contro i server di ricerca che indicizzano tutte le proprietà immobiliari alla ricerca di nuovi risultati. Questo componente non è molto scalabile e presupponendo un aumento della base degli utenti in futuro aumenterebbe troppo il carico di lavoro sui server di ricerca rallentandone quindi la velocità di risposta. Ancora una volta l'architettura modulare e la progettazione del nuovo componente garantirebbe un facile adattamento del nuovo sistema a svolgere questo ulteriore servizio. In questo caso, sarebbe necessario solamente cambiare la frequenza con cui il nuovo componente elabora le informazioni e invia le notifiche. Il maggiore vantaggio dell'utilizzo del nuovo componente rispetto al vecchio servizio è che esso non dovrebbe contattare i

server di ricerca che contengono le informazioni sulle proprietà immobiliari poichè tramite la *prospective search* sono le ricerche salvate dagli utenti ad essere indicizzate in una diversa locazione.

Il fatto che l'implementazione delle nuove feature e il progetto nel suo complesso utilizzi tecnologie *Open Source* offre scenari futuri interessanti per l'evoluzione dei temi trattati in questa tesi. Il lavoro effettuato nel design e nell'implementazione della *prospective search* su piattaforma Apache Solr verrá infatti rilasciato su licenza *Open Source*. Una versione piú generalizzata della soluzione verrá pubblicata su un repository pubblico online per permettere ad altri utenti interessati di contribuire ad estendere e migliorare l'implementazione con l'obiettivo di rendere possibile l'utilizzo di questa nuova feature nella versione ufficiale successiva di Apache Solr.

La tesi, che è stata svolta presso **Trulia** a San Francisco in California, mi ha permesso di apprendere e di confrontarmi con il mondo del software development. Innanzitutto questa esperienza mi ha fornito la possibilità di poter contribuire alla realizzazione di un progetto di innovazione il cui impatto potrebbe avere ripercussioni sulla stessa azienda. Inoltre mi ha permesso di lavorare in un team variegato sia per nazionalità che per background di conoscenze, garantendomi la possibilità di apprendere nuove tecnologie ed allo stesso tempo rinforzare il bagaglio di conoscenze sui sistemi distribuiti e l'ingegneria del software. Non è poi da trascurare l'importanza dell'esperienza di poter vivere e rapportarsi con una cultura differente, il tutto nella località fulcro dell'innovazione tecnologica: la *Silicon Valley*.

Conclusions and future developments

This thesis has presented a work of design and implementation of a real-time notification system that offers the possibility to send a stream of information via email to every registered user to the service through **Trulia.com**.

The project had the ambition to introduce a new feature and integrated it into the current notification system of the company. An environment so complex and hectic as the U.S. housing market needs for a continuous interconnection between the actors of the system such as users and agents or brokers.

The application retrieves new information on real estate properties in full autonomy and after identifing which of these information matches to the user's interests sends an email to them. A User registers his interest directly on the website, after using the normal search. All the interests are then stored and indexed for quick consultation by the component using the Apache Solr search platform. Each time the application incorporates new information on a real estate property, the component executes a filtering logic operation on all interest previously indexed and understands which of these correspond to the new information. A reverse search method called *prospective search* has been implemented on the top of Apache Solr platform that was not natively supported.

To ensure decoupling between all components of the application a hybrid version of the pattern *publish/subscribe* messaging architecture has been used. An additional layer above the implementation has been carried out through the use of JMS APIs on top of Apache ActiveMQ MOM.

During development of the project a great importance was given to try to integrate different technologies to ensure innovation while maintaining stability and security in the enterprise environment. The use of Spring Framework and the Application Container Jetty has brought a great advantage. Both have standardized the development of the application providing the possibility of future extensions.

The work also includes an interface accessible via browser that provides the ability to control the entire application life cycle and set a series of configuration parameters that allows to change the behavior of the system. The control of the application

life cycle is carried through to the use of automated tools like Nagios and the use of targeted logging with Apache Lo4J. At any time of the operation of the component is possible to understand the content of information processed.

Currently the project is in advanced testing phase, once completed this period will be available to the company to be released in production and available to users.

Although the project has been integrated with the current system such as the notification process, it has developed a new architecture that may offer interesting options for future development. A perfect example could be the use of mobile devices as a platform for push real-time notification. The company has already launched in the course of the last year few mobile applications for iPhone, Android, Kindle Fire, and may therefore already have the means to communicate with mobile users easily. Thanks to the decoupled component and the use of messaging queues it is easy to think of the possibility of expanding this service in the world of mobile devices allowing a further integration. Further developments concern the possible future replacement of the current daily and weekly notification system with the component that has been analyzed in this thesis. The current service that performs this task is a batch process that simply executes (daily or weekly) all searches saved by users against index servers that store all real estate properties information and looks for new updates. This component is not very scalable assuming an increase in user base in the future and at the same time it could increase the workload on the search server slowing down the performance of responses on the search servers. Once again, the modular architecture and design of the new component would provide an easy adaptation of the new system to carry out this additional service. In this case, it would only need to change the frequency with which the new component processes the information and sends notifications. The major advantage of using the new component compared to the old service is that the new application does not contact the search servers that provide information about real estate properties because of the use of the *prospective search*. The only information needed to the new component are the user saved searches that are indexed in a different location.

The fact that the implementation of the new features and the overall project uses *Open Source* technology opens interesting future scenarios for the evolution of the topics discussed in this thesis. The work done for designing and implement the *prospective search* on Apache Solr will be released under the open source licence. A more general solution will be published on an online public repository to allow other interested users to help extend and improve the implementation. The next goal of these component will be the future integration with the next ufficial version of Apache Solr.

The thesis, written after a 6 months intership at **Trulia.com** in San Francisco, California, has allowed me to learn and to confront myself with the world of software development. First, this experience has provided me the opportunity to contribute to the creation of an innovation project whose impact could have repercussions on the same company. It also allowed me to work in a variegate team formed with people from different countries and with different knowledge background. That helped me and gave me the the opportunity to learn new technologies and at the same time strengthen the wealth of knowledge on distributed systems and software engineering. Also, it has not to be overlooked the importance of the experience of being able to live and confront with a different culture, all in the center of technological innovation: the *Silicon Valley*.

Ringraziamenti

Vorrei ringraziare in primo luogo il prof. Cesare Stefanelli che ha reso possibile la mia partenza per la California e mi ha dato la possibilità di vivere questa importante esperienza a livello personale e lavorativo. Una grande menzione va data a tutta l'azienda Trulia.com, al Data Acquisition Team e in particolare all' ing. Daniele Farnedi che mi ha assistito e aiutato ad ambientarmi in una realtà di lavoro nuovo e stimolante, lasciandomi un'ottimo ricordo di questa esperienza e facendomi sentire a casa nonostante la distanza. Ringrazio inoltre la mia famiglia che mi ha supportato in questi anni di duro studio e mi ha sempre permesso di inseguire e di reallizzare i miei sogni.

Acknowledgements

First, i would like to thank Professor Cesare Stefanelli that made possible my thesis in California, and has given me the chance to live this experience so important for myself and my future career. A big mention must be given to the entire company Trulia.com, the Data Acquisition Team, and in particular to the engineer Daniele Farnedi who assisted me and helped me settle down into a new and challenging work environment. Thanks to him I have only good memories of this experience, and he always made me feel at home despite the distance from my real home. I also thank my family who supported me in this years of hard study and have always helped me to chase and realize my dreams.

Bibliography

[1] Internet Usage Statistics http://internetworldstats.com/stats.htm

[2] 2011 National Association of REALTORS Profile of Home Buyers and Sellers

[3] One-Stop Search, Forbes.com, January 18, 2008

[4] 2011 REALTOR Technology Survey

[5] JSR914 - JMS Spec http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html

[6] Apache Solr: http://lucene.apache.org/solr/

[7] Apache Tomcat: http://tomcat.apache.org/

[8] Spring Framework Reference: http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/

[9] Jetty Webserver: http://jetty.codehaus.org/jetty/

[10] Apache Maven: http://maven.apache.org/

[11] Nagios http://www.nagios.org/

[12] Standard ECMA-262 3rd Edition - December 1999

Appendix A

Nomenclature

MOM Message-Oriented Middleware

API Application Programming Interface

JMS Java Message Service

TTL Time To Live

JINDI Java Naming and Directory Interface

PS Prospective Search

PSI Prospective Search Index

QoS Quality of Service

DIH Data Import Handler

URL Uniform Resource Locator

QDG Query Document Generator

URLP URL Parser

PD Property Document

<K,V> Pair Key,Value

JSON JavaScript Object Notation