



POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

---

ANALISI, PROGETTO ED IMPLEMENTAZIONE DI  
UNA PIATTAFORMA DI RICERCA ASSISTITA IN  
UN CONTESTO ASSICURATIVO

---

*Laureando:*

Orlando STRIANI

*Relatore:*

Prof. Elio PICCOLO

ANNO ACCADEMICO 2017/2018

Ai miei affetti più cari

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Letteratura e stato dell'arte</b>	<b>3</b>
2.1	L'Information Retrieval e il bisogno informativo dell'utente . .	3
2.1.1	Criteri di classificazione di un sistema . . . . .	4
2.1.2	Architettura tipica e caratterizzazione . . . . .	5
2.1.3	Modelli di Information Retrieval . . . . .	8
2.1.4	Analogie e differenze con i database relazionali . . . . .	10
2.1.5	Performance evaluation di un sistema di IR . . . . .	12
2.1.6	Tecniche avanzate . . . . .	14
2.1.7	Apache Lucene . . . . .	17
2.2	Agenti intelligenti e assistenti digitali . . . . .	20
2.2.1	NLP: cosa si cela dietro ad un Chatbot . . . . .	21
2.2.2	Classificazione degli agenti conversazionali . . . . .	21
2.2.3	Bot economy: ecco come i chatbots daranno una svolta all'economia . . . . .	27
<b>3</b>	<b>Descrizione della soluzione</b>	<b>29</b>
3.1	Presentazione generale dell'architettura . . . . .	29
3.2	Fonte dei dati trattati . . . . .	31
3.3	Indicizzazione e ricerca dei documenti . . . . .	33
3.3.1	Watson Explorer Engine . . . . .	36
3.3.2	BigIndex API . . . . .	40
3.3.3	ZooKeeper . . . . .	42
3.4	Piattaforma assistita di ricerca . . . . .	43
3.4.1	IBM Watson Conversation . . . . .	43
3.4.2	Node-RED . . . . .	45
<b>4</b>	<b>Implementazione del sistema</b>	<b>47</b>
4.1	Generazione e mantenimento degli indici . . . . .	47
4.1.1	Configurazione Watson Explorer Engine . . . . .	48

## INDICE

4.1.2	Initial Load dei dati mediante BigIndex API . . . . .	51
4.1.3	Aggiornamento Near-Online (NOL) dell'indice . . . . .	62
4.2	RESTful Search Layer . . . . .	66
4.3	Piattaforma di ricerca assistita . . . . .	72
4.3.1	Portale di ricerca dei clienti . . . . .	72
4.3.2	Integrazione dell'assistente digitale . . . . .	75
<b>5</b>	<b>Conclusione e Sviluppi futuri</b>	<b>80</b>
5.1	Rivisitazione del modello dei dati e delega dell'attività di ricerca all'assistente digitale . . . . .	81
5.2	Integrazione di nuovi servizi nella piattaforma . . . . .	83
5.2.1	Autenticazione e differenziazione degli utenti . . . . .	83
5.2.2	Continua evoluzione dell'attività di assistenza . . . . .	85

## Elenco delle figure

2.1	Architettura generica e semplificata di un sistema di IR . . . . .	6
2.2	Relazione tra precision e recall . . . . .	13
2.3	Applicazione del LSI nel motore di ricerca Google . . . . .	16
2.4	Step seguiti nello sviluppo di un'applicazione con Lucene . . . .	17
2.5	Machine Learning e Deep Learning al servizio dell'IA . . . . .	20
2.6	Esempio di conversazione con il chatbot Eliza . . . . .	22
2.7	Esempio standard di applicazione di un modello StoS . . . . .	23
2.8	Intention-based agent: identificazione dell'intento . . . . .	24
2.9	Intention-based agent: identificazione dei dettagli . . . . .	24
2.10	Dimostrazione di utilizzo del framework RavenClaw . . . . .	25
3.1	Architettura del sistema . . . . .	30
3.2	Modello relazionale dei dati . . . . .	32
3.3	Document-view relativa ad un cliente . . . . .	34
3.4	Architettura IBM Watson Explorer . . . . .	37
3.5	Architettura fisica di ZooKeeper . . . . .	42
3.6	Architettura generale di una soluzione che sfrutta il Conversation	44
3.7	Web-Interface di Node-Red per la creazione dei flussi . . . . .	46
4.1	Creazione di una search collection su WEX . . . . .	48
4.2	Configurazione di un seed - WEX Search collection . . . . .	49
4.3	Configurazione di un converter - WEX Search collection . . . . .	50
4.4	Configurazione fast-index e term-expansion . . . . .	50
4.5	Configurazione dell'ordinamento dei documenti . . . . .	51
4.6	'Live status' relativo all'indicizzazione . . . . .	62
4.7	Interfaccia web per il CRUD e la simulazione della coda . . . . .	63
4.8	Servizi esposti dal Search Layer . . . . .	67
4.9	Parametri richiesti dai servizi del SL . . . . .	68
4.10	Informazioni sul cliente ricercabili dall'utente finale . . . . .	68
4.11	Filtri applicabili alla ricerca . . . . .	69
4.12	Servizi esposti dal Search Layer . . . . .	70

## ELENCO DELLE FIGURE

4.13	Formato del JSON restituito da un servizio in corrispondenza di una ricerca . . . . .	71
4.14	Form compilabile nel portale di ricerca . . . . .	72
4.15	Esempio di ricerca sul portale . . . . .	74
4.16	Assistente digitale a supporto dell'utente nella ricerca . . . . .	75
4.17	Parte del flusso di dialogo definito nel Conversation . . . . .	77
4.18	Flusso Node-Red utilizzato per la gestione del chatbot . . . . .	78
5.1	Struttura Json di risposta alla API Conversation 'Send message'	86

# Elenco delle tabelle

2.1	DBMS e IRS: sintesi delle differenze . . . . .	11
2.2	Precision e Recall: valutazione della rilevanza in un sistema di IR	12
4.1	Proprietà assegnabili ad un campo nella costruzione del record BigIndex . . . . .	53
4.2	Meccanismo di notifica eventi per la consistenza dei dati . . . .	65
4.3	Intenti ed entità definiti nel modello del Conversation . . . . .	76
4.4	Servizio di chatHistory per tenere traccia di una conversazione	79
5.1	Differenziazione degli utenti sulla piattaforma di ricerca . . . .	83
5.2	Rilascio del feedback utente per l'assistente digitale . . . . .	85

# Capitolo 1

## Introduzione

La piattaforma di ricerca assistita che è stata costruita trae spunto da un progetto reale che l'azienda (Blue Reply s.r.l.), presso la quale è stato svolto il presente lavoro di tesi, ha realizzato per conto di una Compagnia assicurativa multiramo. L'esigenza del cliente era dettata dall'aver modo di offrire all'utente un portale di ricerca delle informazioni nel contesto delle agenzie della Compagnia di assicurazioni sparse in tutto il territorio italiano. Il requisito richiesto era essenzialmente quello di definire un'attività di ricerca full-text, ossia svincolata dalle rigide regole caratterizzanti l'interazione con una collezione di dati strutturati quale un database relazionale. Si è ritenuto tale progetto essere un valido spunto da cui partire facendo propria l'idea alla base, ovvero la predisposizione di un *portale di ricerca* previa indicizzazione dei dati e lo sviluppo di una soluzione propria che integrasse opportunamente gli strumenti approfonditi durante il periodo di attività di stage in azienda e che permettesse, alla soluzione stessa, di presentarsi come un prodotto finale completo e autoconsistente. La soluzione spazia quindi dal reperimento dei dati strutturati da un database assicurativo all'indicizzazione elaborata di questi per l'attività di ricerca, includendo un meccanismo a garanzia della consistenza e integrità dei documenti indicizzati, il tutto con il fine ultimo di offrire all'utente finale una piattaforma di ricerca semplice ed intuitiva, ma al contempo esaustiva. La ricerca fruibile dall'utente è inoltre guidata da un *assistente digitale* integrato nella piattaforma che, opportunamente addestrato, interpreta le richieste espresse in linguaggio naturale e dialoga con l'interlocutore con lo scopo di fornirgli assistenza e fugare ogni suo dubbio relativo all'attività di ricerca sul portale.

Il presupposto che costituirà la forza trainante del lavoro di tesi presentato è quello di facilitare l'accesso alle informazioni cui un utente è interessato. Il tradizionale utente non esegue una ricerca in uno spazio di scoperta in cui non



sa se il contenuto è presente oppure no, bensì in uno spazio in cui è pressoché certo che il contenuto sia presente e che lui lo debba “solo ed esclusivamente trovare”, ed è individuabile un chiaro legame tra facilità di reperimento dell’informazione cercata e la soddisfazione dell’utente, che si tramuta tanto in incremento dei ricavi quanto in minor propensione all’abbandono del servizio.

Alla base dell’architettura proposta è presente il concetto di *motore di ricerca*, che rappresenta un esempio concreto e con il quale ci cimentiamo quotidianamente, di applicativo frutto degli studi di **Information Retrieval**. Il tema dell’Information Retrieval (letteralmente ‘recupero delle informazioni’) è un tema che in particolar modo negli anni più recenti, grazie all’avvento della rivoluzione tecnologica (in primis informatica e di Internet) e soprattutto grazie alla smisurata crescita della mole di informazione digitale immagazzinata, è divenuto sempre più caldo, riuscendo a trovare nuova vita e una nuova, diversa dimensione applicativa nel campo dell’informatica. L’Information Retrieval nasce dall’incrocio di molteplici discipline, che trovano nell’IR il mezzo comune tramite cui soddisfare il cosiddetto bisogno informativo dell’utente.

Il lavoro di tesi è strutturato come segue: il primo capitolo illustra la letteratura e lo stato dell’arte dei temi cardine legati all’Information Retrieval (IR) e all’elaborazione del linguaggio naturale (**Natural Language Processing**) negli agenti conversazionali. Segue un capitolo dedicato alla presentazione della soluzione realizzata e alla descrizione dell’architettura e delle componenti utilizzate. Il terzo capitolo affronta nel dettaglio l’implementazione della soluzione descritta, mentre nell’ultimo capitolo saranno fatte le dovute considerazioni sul risultato raggiunto e si aprirà ad eventuali sviluppi futuri.

# Capitolo 2

## Letteratura e stato dell'arte

*Seppur intorno agli anni '90 alcuni studi avessero appurato il fatto che la maggior parte della gente preferisse ancora fruire dell'informazione direttamente dall'umano piuttosto che per tramite di un sistema di Information Retrieval, diversamente, da un ventennio a questa parte, l'ottimizzazione e l'evoluzione raggiunta dall'IR han fatto sì che il motore di ricerca divenisse uno standard nell'ambito dell'information finding e che un sempre più ampio bacino di utenza iniziasse a prenderci confidenza e a preferirlo, per immediatezza ed efficacia.*

*Lo stato dell'arte presentato qui di seguito descriverà le principali problematiche teoriche, metodologiche e implementative dei sistemi di information retrieval che influenzano la progettazione e la realizzazione dei motori di ricerca. Completa il capitolo, infine, lo stato dell'arte relativo alle tecniche di Natural Language Processing nel contesto di un agente intelligente.*

### **2.1 L'Information Retrieval e il bisogno informativo dell'utente**

Il termine Information Retrieval è stato coniato verso la fine degli anni '40 da Calvin Mooers, padre di uno dei primi linguaggi di programmazione (il TRAC), in un contesto a metà tra l'informatica e la biblioteconomia, ossia la scienza che si occupa della catalogazione di grossi archivi attraverso metodi che assegnino ai vari elementi presenti degli identificativi univoci.

Il significato del termine Information Retrieval è di ampia interpretazione nel linguaggio comune ma, volendone assegnargli una definizione tecnica intendendo l'IR alla stregua di un campo di studio accademico, è possibile descrivere l'IR come l'insieme delle tecniche utilizzate per il recupero mirato dell'informazione in formato elettronico (documenti, metadati, file presenti

all'interno di banche dati o nel www).

[1, capther 1, p. 1]:

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Il settore dell'Information Retrieval è stato studiato fin dagli anni '70, sebbene già da un ventennio l'espressione iniziasse prepotentemente a prender piede (il primo computer studiato per l'IR risale al 1952). La crescita vertiginosa d'interesse nei confronti del nuovo settore si ebbe negli anni '90, grazie all'esplosione del web.

### 2.1.1 Criteri di classificazione di un sistema

**Natura e tipologia dei dati** Una prima chiave utilizzata nella classificazione di un sistema di IR è rappresentata dalla natura e dal tipo dei dati. Sebbene, in accordo alla definizione poc'anzi riportata, l'IR sia nata per far fronte al trattamento dei dati *non strutturati* (tipico esempio i file contenenti testi oppure un file multimediale come un file audio o video), l'osservazione che può essere fatta è la seguente: in realtà, la percentuale di dati effettivamente non strutturati è piuttosto bassa. Basti pensare al fatto che la maggior parte dei testi digitali, oggi, presentano una struttura definita (intestazione, paragrafi, note, etc.), o rappresentata nei documenti per mezzo di un esplicito markup (ci si riferisce, in particolare, a documenti XML). La categoria di dati appena citata cade sotto il nome di *semi-strutturata* (si parla allora di Structured Information Retrieval, o SIR), e si distingue dalle restanti due tipologie per il fatto che nonostante non vi sia alcun limite strutturale all'inserimento dei dati, le informazioni vengono, comunque, organizzate secondo delle logiche strutturate e interoperabili.[2]

**Scala e raggio d'azione** Un'ulteriore distinzione tra i sistemi di IR è basata sulla dimensione applicativa che caratterizza il sistema stesso. Una prima dimensione è sicuramente quella della rinomata **web-search**, dove l'enorme mole dei documenti da indicizzare e la necessità di gestire i diversi e particolari aspetti del web portano ad uno studio attento nella progettazione del sistema. Classico esempio quello relativo al rendere vano il tentativo di un gestore di un sito web di manipolare il contenuto di una pagina con l'intento

di migliorarne il posizionamento tra i risultati di un motore di ricerca. All'estremo opposto della dimensione web-search figura la **personal-information-retrieval**, una dimensione più contenuta rispetto alla prima, ma allo stesso modo di spessore e certamente attuale. In particolar modo negli ultimi anni gli applicativi di posta elettronica hanno iniziato a fornire all'utenza non più solo la feature di ricerca, ma anche quella della classificazione del testo. Comunemente, una serie di strumenti per la classificazione (manuale o automatica) delle mails sono forniti all'utente, di modo che questo possa aver modo di disporle entro specifiche cartelle. Questo va aggiunto alla capacità di filtro dello spam (junk mail), resa disponibile oramai da quasi tutti i programmi di questo tipo. Problematiche riscontrabili nella dimensione applicativa appena descritta sono la gestione della grande varietà di tipologie di documenti presenti su di un classico personal computer, ma anche, e soprattutto, la necessità di garantire un sistema di ricerca che sia poco (o per niente) oneroso per chi ne fa utilizzo, sia in termini di costi sia in termini di requisiti richiesti (startup, processing, e spazio su disco).

Ultimo, ma non per importanza, lo spazio di **enterprise,institutional, and domain-specific search** che, come scala di applicazione, trova posto tra le due precedentemente illustrate. Qui, tipicamente, l'informazione è mantenuta in file systems centralizzati o immagazzinata in una o più macchine dedicate, e resa da queste disponibile all'attività di ricerca per tramite di collezioni.

### 2.1.2 Architettura tipica e caratterizzazione

Per il recupero dell'informazione i sistemi di IR fanno affidamento sui linguaggi d'interrogazione basati su comandi testuali. Di fondamentale importanza sono i concetti di **query** e di **oggetto**. La prima è rappresentata generalmente da una stringa composta di parole chiave, o **keywords**, e che rappresenta l'informazione richiesta dall'utente. L'oggetto è, invece, un'entità che racchiude delle informazioni in una banca dati (si pensi ad un documento di testo, ad esempio, che può essere visto alla stregua di un oggetto di dati). Il sistema, a seguito dell'interrogazione, fornisce come output tutti e soli i documenti rilevanti trovati per un dato utente con una data richiesta informativa, ordinati per rilevanza. Se ne evince che, rispetto alla ricerca classica delle basi di dati, l'enfasi non è posta sulla ricerca di dati in quanto tali, ma bensì sulla ricerca di informazioni.

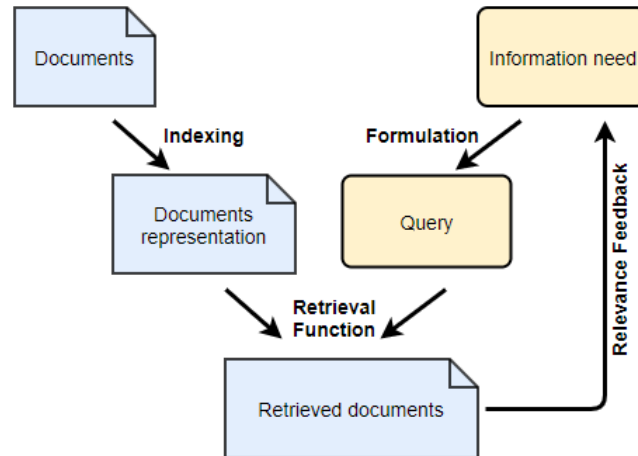


Figura 2.1: Architettura generica e semplificata di un sistema di IR

**Retrieval technique** La tecnica di recupero dell'informazione adottata da un sistema di IR altro non è che il meccanismo, interno al sistema stesso, che lo guida nell'associare ad un documento nella raccolta l'etichetta di rilevante o non rilevante, in rapporto ad una ben specifica interrogazione dell'utente. Le tecniche per il recupero sono essenzialmente di due tipi: per corrispondenza esatta (**exact match**) e per similitudine o corrispondenza parziale (**partial match**). Le prime danno per assunto che la richiesta contenga esattamente le informazioni presenti nella componente di testo del documento. Gli svantaggi di tali tecniche sono evidenti. Enunciamone alcuni:

- gran parte dei documenti rilevanti sono potenzialmente ignorati, se il testo non matcha esattamente con il contenuto dell'interrogazione
- non esiste un ordine per rilevanza rispetto all'interrogazione
- non è possibile in alcun modo tener conto dell'importanza relativa tra i concetti presenti, sia nell'interrogazione che nei documenti
- il linguaggio d'interrogazione si presenta spesso come un linguaggio complicato
- l'efficacia del sistema dipende strettamente dalla misura secondo cui le due rappresentazioni da confrontare (query e documento) siano basate o meno su di un vocabolario comune

Le tecniche di recupero per **partial match**, invece, sono basate sull'assunzione che nella richiesta dell'utente le informazioni possano essere specificate in maniera parziale, e che i documenti restituiti possano essere ordinati per valori crescenti o decrescenti di rilevanza. Tali tecniche sono sicuramente più

flessibili, ed è su queste che oggi giorno è concentrato il maggior sforzo di ricerca.

**Rappresentazione dei documenti** In base alla rappresentazione data ad un documento, un sistema di IR viene categorizzato come adottante un metodo di:

- **Rappresentazione diretta:** il documento è rappresentato dalle parole che lo formano
- **Rappresentazione indiretta:** il documento è rappresentato da termini di indicizzazione (di derivazione manuale o automatica) che ne descrivono in maniera sintetica ma completa il suo contenuto.

Nella *rappresentazione diretta*, ai fini della ricerca, vengono omessi i termini contenuti in una stop-list, o lista di parole da ignorare (preposizioni, congiunzioni, etc.), che sono ritenute di poco conto e non rappresentative del contenuto del documento. Questo tipo di rappresentazione è in generale inadeguata, poichè consente solo ed esclusivamente il recupero di testi per mezzo di richieste che specificano una condizione sui termini presenti.

La *rappresentazione indiretta*, invece, associa un insieme di parole chiave (semplici o composte) ad un testo, in modo che queste siano rappresentative -in modo sintetico- del documento. L'operazione di estrazione delle parole chiave è denominata classificazione o *indicizzazione*, ed è eseguita manualmente da esperti o automaticamente da tecniche ad hoc basate su metodi statistici.

**Indicizzazione** L'indicizzazione è il processo di rappresentazione dei documenti tramite una descrizione sintetica degli stessi. Tipicamente, l'indicizzazione genera un insieme di *termini indice* (solitamente pesati) che vengono usati come base della rappresentazione formale di un documento o di una query di ricerca. Durante la fase di recupero, a seguito di una richiesta effettuata dall'utente, tali termini sono utilizzati al posto del documento come dei *surrogati* per la rappresentazione del documento originale.

L'indice più utilizzato è l'**inverted index**: viene memorizzato l'elenco dei termini contenuti nei documenti della collezione e, per ogni termine, viene mantenuta una lista dei documenti nei quali è presente tale termine. Tale tipologia di indice è valida per query di semplice natura (insiemi di termini), ma è da modificare qualora si vogliano gestire altre tipologie di query più complesse, come nel caso di *phrase match* e *proximity search*.

L'utilizzo degli indici in un sistema di IR semplifica e al contempo accelera

il recupero dei documenti. Il processo di indicizzazione può essere di tipo manuale (migliore perché più semantico, ma soggettivo e costoso) o di tipo automatico (su base statistica e quindi peggiore, ma stabile ed economico).

La prima fase del processo di indicizzazione è quella della **tokenizzazione**, in cui un tokenizer trasforma uno stream di testo in un elenco di token che rappresentano i candidati a diventare le entries dell'indice. Durante questo step vengono applicate trasformazioni quali l'eliminazione delle parole contenenti cifre, la *de-hyphenation* (divisione in più parole ove presente un trattino), la trasformazione delle maiuscole in minuscole, e l'eliminazione della punteggiatura.

Il numero di termini soggetti ad indicizzazione viene ridotto utilizzando dei **moduli linguistici**, che effettuano operazioni quali l'eliminazione delle *stop-words*, lo *stemming* (riduzione delle parole alla loro radice, rimuovendo prefissi e suffissi), i *thesauri* (gestione dei sinonimi tramite classi di equivalenza predefinite), e la *lemmatization* (riduzione delle parole alla loro radice grammaticale).

Dal punto di vista della mole di indice generato e della precisione della ricerca, tali tecniche sono senza alcun dubbio benefiche, ma non sempre però il risultato ottenuto in termini di qualità della risposta è migliore, e si tenga presente che la trasformazione del testo rende più difficile la ricerca all'utente.

### 2.1.3 Modelli di Information Retrieval

La categorizzazione dei sistemi di Information Retrieval viene eseguita astraendo le caratteristiche salienti che sono alla base dei sistemi stessi:

- lo stile di rappresentazione dei documenti, ovvero l'insieme delle possibili chiavi di accesso al documento
- lo stile di rappresentazione delle richieste, in termini di esemplificazione delle domande formulabili dall'utente
- la regola di recupero, ossia la modalità di confronto tra rappresentazione di documenti e di richieste
- l'insieme degli indicatori di valore informativo associate ai documenti

I due modelli classici nell'Information Retrieval sono il modello **booleano** e il modello **vettoriale**. Non sono però gli unici modelli presenti, dal momento che ve ne sono di intermedi come, su tutti, il modello *fuzzy* e quello *probabilistico*.

**Modello booleano** Il modello booleano risale agli anni '50 (storicamente, è stato il primo) ed è ancora utilizzato nei sistemi industriali e nei motori di ricerca su documenti nel Web. È un modello a corrispondenza esatta, il più semplice tra quelli disponibili nello stato dell'arte. Si basa sulla teoria degli insiemi e sull'algebra booleana.

In questo modello i documenti vengono rappresentati per mezzo di termini (*keywords*), scelti durante la fase di indicizzazione, che ne rappresentano il contenuto. Le query vengono specificate come espressioni dell'algebra booleana basate sulle keywords, ovvero come un elenco di termini interconnessi dagli operatori di AND, OR, e NOT. La strategia di ricerca adottata, infine, è basta sul criterio di decisione binario e non viene tenuta in considerazione la nozione di grado di rilevanza (*rilevante -1-* e *non rilevante -0-* sono i due unici stati associati ad un documento in fase di ricerca). Non esiste, dunque, un controllo sul numero dei documenti ritornati, in quanto vengono restituiti tutti e soli i documenti che matchano in maniera esatta la query, senza alcun ordinamento e quindi la possibilità di avere un *relevance feedback*.

La semantica risulta di semplice comprensione, ed il formalismo adottato è semplice e chiaro. Il modello appena descritto risulta essere efficace in ambienti controllati e fruito da utenti bene addestrati: è previsto un addestramento dell'utente, che deve sapere cosa poter chiedere e come chiederlo.

**Modello vettoriale** Nato intorno agli anni '60, il modello vettoriale è un modello utilizzato nei sistemi industriali e, originariamente, dai motori di ricerca Web. È un modello a corrispondenza parziale e nasce per venire incontro alle limitazioni del criterio di valutazione della rilevanza del modello booleano. I documenti sono rappresentati per tramite di un *vettore di numeri* (0 se il termine non è presente, altrimenti un numero pari al peso del termine nel documento) di lunghezza pari al numero di termini utilizzati per rappresentare il documento stesso. L'interrogazione è espressa mediante un insieme di termini, mentre la sua rappresentazione equivale ad un vettore (simile al vettore-documento) con una gran serie di 0 e qualche 1 in corrispondenza ai termini specificati dall'utente nella richiesta. Un modello specifico appartenente a questa classe si distingue dagli altri in base alle scelte di pesatura dei termini nelle interrogazioni, di pesatura nei termini nei documenti, della metrica adottata (*funzione di similarità*, che calcola il grado di similitudine tra due vettori e che banalmente potrebbe coincidere con la distanza degli spazi lineari), e del criterio di accettazione (*soglia di rilevanza*).

L'approccio vettoriale presenta dei problemi legati al fatto che sul calcolo del-



la rilevanza incidano sia la lunghezza che il numero dei documenti, oltre al fatto che non viene considerato l'ordine relativo tra i termini.

#### 2.1.4 Analogie e differenze con i database relazionali

L'Information Retrieval, attualmente, gioca sempre più un ruolo dominante nell'accesso all'informazione, superando di gran lunga la ricerca *database-style*. I motori di ricerca **full-text** hanno una storia relativamente giovane, se si considera come termine di confronto quella di un tradizionale *database engine*. L'esigenza di adottare un approccio differente nella ricerca iniziò a divenire impellente nel momento in cui si resero evidenti i limiti dell'allora attuale soluzione.[3]

Nei DBMS le informazioni vengono rappresentate come insiemi di dati strutturati e relazioni tra entità, mentre nei sistemi di IR l'informazione è rappresentata come insiemi di testi. Per quanto concerne le richieste, invece, in un sistema di Information Retrieval (IRS) sono espressioni imprecise del bisogno informativo dell'utente (soggettive e/o incomplete). La risposta fornita dal sistema ad una richiesta, infine, varia decisamente nei due diversi contesti: in un DBMS il risultato restituito è quello che soddisfa in maniera esatta la condizione della query di ricerca, mentre in un IRS è l'utente a stabilire quali dei documenti restituiti siano davvero rilevanti per il suo bisogno informativo e, soprattutto, il risultato è in forma di documento che *potrebbe* contenere la risposta, piuttosto che in forma di risposta diretta. Come già detto più volte, il concetto di rilevanza è centrale nell'Information Retrieval, e lo scopo fondamentale di una strategia automatica di retrieval è quello di restituire il più alto numero di documenti rilevanti (e, allo stesso tempo, il minor numero possibile di documenti non rilevanti).

L'uso dei database relazionali presenta una serie di problemi non indifferenti: il primo sta nel fatto che, anche per richieste al sistema relativamente semplici, le interrogazioni SQL che ne derivano possono divenire molto complicate; per questa ragione il linguaggio SQL risulta poco adatto in un'interfaccia utente, come linguaggio di interrogazione a disposizione degli utilizzatori del sistema. Dovrà esser previsto un qualche tipo di *parser* in grado di trasformare le richieste dell'utente, espresse in un linguaggio ad esso più congeniale, nei corrispondenti statement di interrogazione utilizzati da un motore di database. Il secondo inconveniente è legato al fatto che le relazioni debbano essere in 1NF (prima forma normale): se si sceglie di utilizzare un database relazionale è necessario decidere quale debba essere l'unità atomica di informazione da registrare in ogni singola tupla. La normalizzazione delle relazioni, se ap-

plicata ad un sistema di IR (avendo ad esempio scelto di associare ad ogni tupla una singola parola del testo), equivarrebbe ad una perdita di efficienza. Le query di interrogazione risulterebbero alquanto complesse da scrivere e richiederebbero una lunga serie di join per poter essere valutate (join che rallenterebbero inesorabilmente il processo di ricerca).

	DBMS	IRS
Tipologia dei dati	Strutturati	Testo
Richiesta	Completa e precisa	Incompleta e vaga
Criterio di scelta	Corrispondenza esatta	Corrispondenza parziale
Risultato	Dati richiesti	Documenti probabilmente rilevanti

Tabella 2.1: DBMS e IRS: sintesi delle differenze

I due campi dei database e dell'IR si sono evoluti separatamente per un lungo periodo ma, con l'andare del tempo, si è arrivati ad una sempre maggiore *integrazione* tra i due. Le ragioni alla base di tale processo sono essenzialmente di natura:

- funzionale, perchè sempre più applicazioni oramai richiedono di poter accedere ad una combinazione di dati formattati e non
- tecnologica, in quanto un DBMS offre un completo supporto per la risoluzione di una serie di problemi che un sistema di IR (come del resto una qualsiasi altra applicazione) deve affrontare, ovvero la gestione della *concorrenza*, la *recovery*, l'*indexing* dei dati, e il *parallel processing*

Gli approcci principalmente seguiti nell'attività di integrazione prevedono di:

- combinare i sistemi di IR e RDBMS esistenti in modo che i dati strutturati vengano immagazzinati nel database relazionale, mentre quelli non strutturati vengano mantenuti nel sistema di IR; bisogna però creare un'interfaccia in grado di estrarre dalle query le richieste di dati per ogni sottosistema, indirizzarle e combinare poi le risposte provenienti dalle singole parti
- modificare un sistema di IR per far sì che sia in grado di gestire query relazionali, oppure estendere un RDBMS in modo che possa trattare query su oggetti di tipo testuale
- implementare un sistema di IR come applicazione di un database relazionale (una soluzione adottata è quella di rappresentare l'inverted index associato ad una collezione di documenti mediante un set di tabelle)

### 2.1.5 Performance evaluation di un sistema di IR

Un tradizionale sistema di Data Retrieval può essere valutato tenendo conto di svariate misure:

- Velocità d'indicizzazione (numero di documenti indicizzati nell'arco di un'ora)
- Velocità di ricerca (in funzione della dimensione dell'indice)
- Espressività del linguaggio di interrogazione

La performance evaluation è dunque misurabile ma la più importante misura è quasi certamente il grado di soddisfazione dell'utente (*retrieval performance evaluation*). Quest'ultimo è un parametro che dipende fortemente dal meccanismo **ranking-based** presente alla base di un sistema. Si può misurare la soddisfazione di un utente in termini di velocità di ottenimento del risultato, ma è chiaro che una risposta istantanea ma inutile non sarà di sicuro gradita. Tale parametro non risulta quindi di facile misurazione, e dipende sia dal tipo di utente che dall'applicazione.

	Rilavanti	Non rilevanti
Restituiti	TP (True positive)	FP (False positive)
Non restituiti	FN (False negative)	TN (True negative)

Tabella 2.2: Precision e Recall: valutazione della rilevanza in un sistema di IR

In generale, il miglior modo per valutare un sistema di IR è quello di considerare la rilevanza dei risultati, tenendo bene a mente che l'obiettivo primario dell'IR è quello di rendere al contempo rapida, semplice e valida la ricerca dei documenti all'interno di una collezione di dati: la misura di quanto ciò avvenga in maniera ottimale è data per mezzo dei due parametri di *effectiveness* ed *efficiency* [4]. Il primo parametro coincide con il grado di soddisfazione del bisogno informativo dell'utente da parte del sistema, ed è quindi strettamente correlato alla rilevanza della risposta rispetto alla richiesta fornita in input. Il parametro di *efficiency*, invece, equivale al quantitativo di risorse consumate dal sistema (CPU, memoria, tempi di risposta, etc.), ed è legato dunque alla potenza computazionale della macchina utilizzata e alla scelta degli algoritmi utilizzati nella ricerca.

Per quanto concerne l'*effectiveness* di un sistema di IR, sono due i fattori che entrano in gioco nella sua valutazione: **precision** e **recall** (definiti da

Perry, Kent & Berry, nel 1955). La prima esplica la proporzione di documenti effettivamente rilevanti in rapporto al totale dei documenti restituiti dal sistema.

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|}$$

Il parametro del recall corrisponde invece alla proporzione dei documenti rilevanti restituiti dal sistema in rapporto al totale dei documenti rilevanti (restituiti e non) presenti nella collezione in uso.

$$recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|}$$

Queste due misure sono solitamente usate assieme, in combinazione, nel calcolo di quello che prende il nome di **F1 Score** (o f-measure), utilizzato per fornire una unica misura per un dato sistema. Tale misura costituisce la media pesata di precision e recall ed è data dalla seguente formula[5]:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

In linea teorica, un sistema di IR perfetto è un sistema ove l'insieme dei documenti rilevanti coincide esattamente con quello dei documenti restituiti, ovvero un sistema caratterizzato da precision e recall pari al 100%.

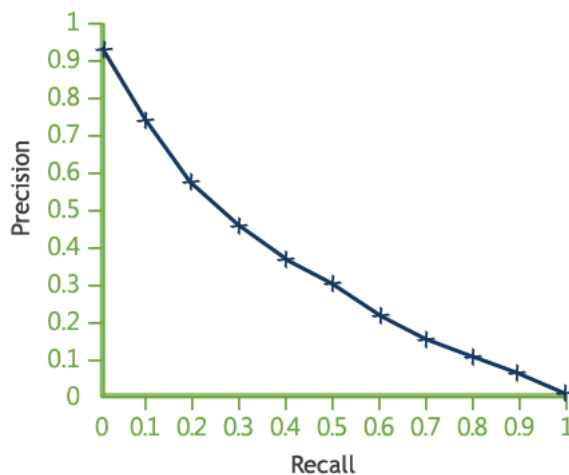


Figura 2.2: Relazione tra precision e recall

Nella pratica, invece, quello che succede è che le due misure siano inversamente proporzionali: spesso ad un'elevata precision corrisponde un basso recall (è probabile che molti documenti rilevanti non siano stati recuperati dal

sistema), mentre ad un alto recall corrisponde solitamente una bassa precision (un gran numero di documenti restituiti, ma pochi dei quali effettivamente rilevanti). Questi sono due effetti negativi che caratterizzano un sistema di IR. Nel gergo, la presenza di documenti non rilevanti fra quelli recuperati viene riconosciuta come *effetto rumore*, mentre al mancato recupero di documenti rilevanti viene assegnato il nome di *effetto silenzio*.

I due parametri di precision e recall possono essere migliorati se, in fase di recupero dell'informazione, si applica il processo noto come retroazione sulla rilevanza o **relevance feedback**[6], secondo il quale l'utente, con l'aiuto del sistema, riformula la richiesta in base ai documenti recuperati in precedenza. È stato dimostrato che il miglioramento nella precisione, rispetto ad un processo di ricerca senza relevance feedback, può arrivare fino al 90%. È bene però sottolineare che la rilevanza di un documento non può essere garantita dal sistema, ma può essere stabilita solo da colui che formula la richiesta. Di conseguenza può accadere che documenti che l'utente considererebbe rilevanti non facciano parte dei documenti recuperati dal sistema, e viceversa. Un sistema di Information Retrieval cerca proprio di limitare questi due inconvenienti che, in generale, non possono essere eliminati.

Il problema appena messo in evidenza risulta dovuto sia alla difficoltà da parte di chi formula la richiesta di caratterizzare in modo univoco, ma al contempo sintetico, il contenuto dei documenti che desidera recuperare, sia alla difficoltà di rappresentare in modo completo il contenuto dei documenti.

### 2.1.6 Tecniche avanzate

Al fine di incrementare l'efficacia di un sistema di IR sono utilizzate diverse tecniche tra le quali il *Ranking probabilistico*, il *Latent semantic Indexing*, e la *Relevance Feedback* in cooperazione con la *Query Expansion*.

**IR probabilistico** L'idea chiave alla base di tale tecnica è la classificazione dei documenti in ordine di probabilità di rilevanza rispetto all'informazione richiesta, ovvero in termini di  $P(\text{rilevante}|\text{documento}_i, \text{query})$ . Viene adottato dunque il modello probabilistico e, nello specifico, il **probability ranking principle** (principio di pesatura probabilistico), che stima la rilevanza di un documento rispetto ad una query di ricerca. Dal momento che il modello probabilistico risulta di difficile stima, si effettuano una serie di passaggi intermedi (l'inversione con Bayes di variabile aleatoria condizionante e condizionata, ad esempio) e di semplificazioni (su tutte, l'indipendenza statistica di certe variabili), in modo da rappresentare il modello iniziale in termini di

probabilità più semplici da stimare su di un campione.

Sia  $d$  un documento appartenente alla collezione, ed  $R$  la rilevanza del documento rispetto ad una specifica query  $q$  ( $R = 1$ ), ed  $NR$  la non-rilevanza ( $R = 0$ ). Si vuole stimare la probabilità  $p(R|d, q)$ , ossia la probabilità che  $d$  sia rilevante, data la query  $q$ . Sulla base del teorema di Bayes:

$$p(R|d, q) = \frac{p(d|R, q)p(R|q)}{p(d|q)}$$

$$p(NR|d, q) = \frac{p(d|NR, q)p(NR|q)}{p(d|q)}$$

dove  $p(R|q)$  e  $p(NR|q)$  indicano le probabilità a priori di recuperare un documento (non) rilevante, e  $p(d|R, q)$  e  $p(d|NR, q)$  la probabilità che, se si trova un documento rilevante (non rilevante), questo sia  $d$ . Ne deriva che  $p(R|d, q) + p(NR|d, q) = 1$ .

Il Probability Ranking Principle (PRP) stabilisce che, su base della cosiddetta *Optimal Decision Rule*,  $d$  è rilevante iff  $p(R|d, q) > p(NR|d, q)$ . Il processo di Information Retrieval, se modellato in termini probabilistici, è caratterizzato dagli eventi aleatori di occorrenza di una query, rilevanza o non rilevanza di un documento, e occorrenza di un termine in un documento. Per il calcolo delle probabilità condizionate, infine, si utilizzano degli *stimatori*: il modello più semplice è il *Binary Independence Retrieval* (BIR), e una valida alternativa è rappresentata dalle **Reti Bayesiane**.

L'obiettivo di una Rete Bayesiana, nel contesto della performance evaluation di un sistema di IR, coincide con lo stimare la probabilità che un documento soddisfi la richiesta (*inferenza*), data una richiesta di informazione da parte dell'utente (*evidenza*). L'approccio adottato modella documenti e bisogno informativo rispettivamente come *document network* e *query network*.

**Latent Semantic Indexing** Diversamente dai metodi tradizionali di ranking che calcolano l'attinenza di un documento ad una query sulla base della presenza o meno di parole contenute nella query, nel Latent Semantic Indexing (LSI) la ricerca avviene per concetti: il concetto non è da intendere come astrazione/generalizzazione di un termine ma bensì come un insieme di termini correlati, anche noti come co-occorrenza o, nel gergo, **dominio semantico**.

LSI è in grado di rilevare, data una collezione di documenti, alcune n-uple di termini che co-occorrono in maniera frequente. È dimostrato che, utilizzando un ranking basato sulla co-occorrenza dei termini, risulta possibile assegnare un miglior ranking ai documenti.

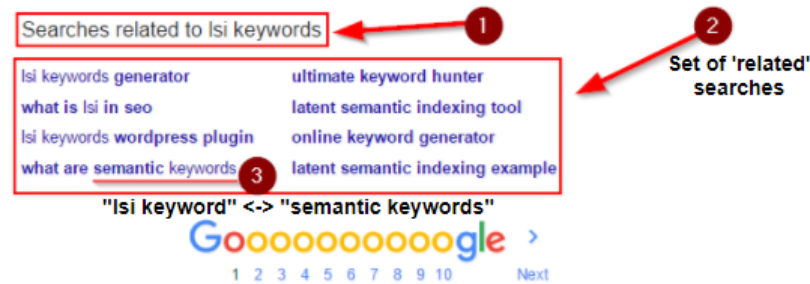


Figura 2.3: Applicazione del LSI nel motore di ricerca Google

**Relevance Feedback e Query Expansion** Relevance Feedback (RF) e Query Expansion (QE) sono delle tecniche utilizzate per migliorare il recall di una query. L'idea alla base del Relevance Feedback è che si richieda all'utente di selezionare i documenti più rilevanti, una volta presentato un set iniziale di documenti rappresentanti il risultato per una data query. Il feedback viene usato per riformulare la query in oggetto, presentando un nuovo set di risultati all'utente, ed eventualmente reiterando il processo.

Nella Query Expansion, invece, vengono aggiunti dei termini oltre quelli iniziali (nuovi termini estratti dai documenti prescelti dall'utente), per far sì che la qualità della ricerca sia migliorata. In alternativa alla QE, è possibile tener conto del feedback dell'utente per tramite della *Term Reweighting*, mediante la quale viene aumentato il peso dei termini presenti nei documenti più rilevanti e diminuito il peso di quelli che non vi compaiono.

Per quanto concerne il Relevance Feedback, non viene utilizzato un feedback esplicito fornito dall'utente (per via della riluttanza caratterizzante il comune utente), ma bensì uno *Pseudo Feedback*: assunto che i primi  $n$  top-ranked siano i documenti più interessanti, la query viene espansa includendo dei termini correlati con i termini della query, servendosi degli  $n$  top-ranked.

Google è un classico esempio d'utilizzo delle tecniche di Query Expansion. Il colosso americano fa uso di strumenti quali:

- Errori di digitazione ('wigedts' diviene 'widgets')
- Traduzione ('organizzazione mondiale sanità' -> 'world health organization')
- Word stemming ('translator' -> 'traduttore, translation')
- Sinonimi (solo se è evidente che il termine sia stato usato in modo improprio)
- Related Search (LSI keywords)

### 2.1.7 Apache Lucene

Lucene[7] è un progetto open source e gratuito Apache Jakarta, che fornisce una serie di API scritte in Java e funzionali all'implementazione di potenti (altamente scalabili ed efficienti) motori di ricerca *full-text* per applicazioni J2EE (Java 2 Enterprise Edition).

Nel dettaglio, l'insieme delle API offerte da Lucene<sup>1</sup> consente di implementare funzionalità di indicizzazione rispetto a differenti formati di documento o ad informazioni estrapolate direttamente da un database. La creazione di un'applicazione di ricerca[8] completa che utilizza Lucene riguarda principalmente gli step di indicizzazione dei dati, di ricerca dei dati e di visualizzazione dei risultati di ricerca.

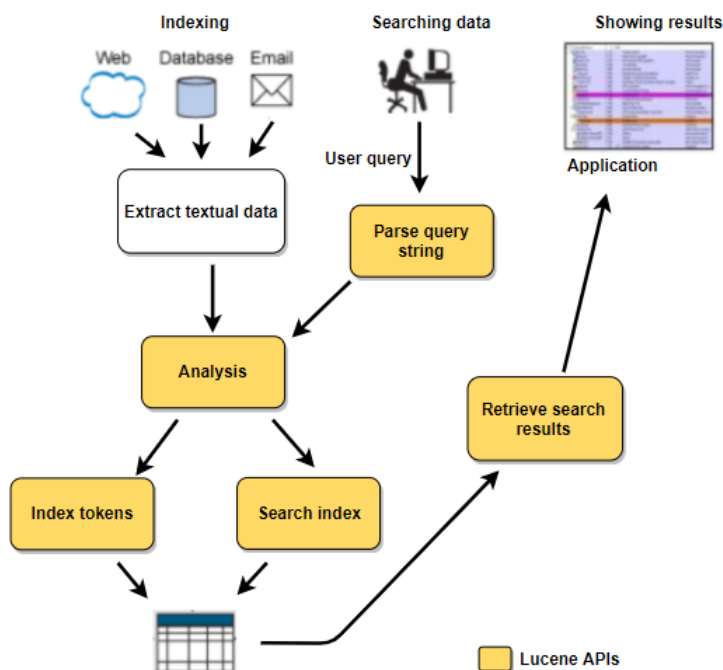


Figura 2.4: Step seguiti nello sviluppo di un'applicazione con Lucene

L'*indicizzazione* è un processo di conversione dei dati di testo in un formato che facilita la ricerca rapida. Lucene memorizza i dati di input in una struttura dati denominata **inverted index**, che viene memorizzato nel file system o nella memoria come un insieme di file di indice. Prima che i dati di testo vengano aggiunti all'indice, vengono elaborati da un *analizzatore* (utilizzando un processo di analisi). Gli analizzatori sono responsabili della pre-elaborazione dei dati di testo e della loro conversione in token memorizzati nell'indice. L'indice di Lucene è interrogabile in fase di ricerca mediante delle query che

<sup>1</sup><https://lucene.apache.org/>



rispettano una logica ben definita e mascherata all'utente finale, al quale è richiesta solo ed esclusivamente una chiave di ricerca rispetto alla quale interrogare l'indice precostruito.

Il risultato di una generica ricerca su di un indice è una lista di documenti (precedentemente indicizzati) che contengono una o più parole costituenti la keyword di ricerca. Ad ogni documento restituito è assegnato uno **score** (punteggio), che indica all'utente quanto il documento in oggetto sia affine rispetto alla chiave di ricerca immessa.

L'indice di Lucene è composto da un insieme di **Document** di Lucene, che risulta a sua volta composto da **Field**, ossia coppie <nome, valore> che caratterizzano e contengono l'informazione mantenuta. Da qui una stretta analogia tra un documento in Lucene e un tradizionale record di una base di dati.

Lucene combina[9] il modello booleano (BM) con quello a spazio vettoriali (VSM): se un documento risulta 'approvato' dal BM, il suo score viene calcolato con il VSM. Il modello booleano viene dunque utilizzato per restringere i documenti che devono essere valutati in base all'uso della logica booleana nella specifica query. In uno step successivo, riprendendo l'idea alla base del VSM, Lucene assegna al documento uno score che è tanto più alto quanto maggiore è il numero di volte in cui il termine di ricerca appare nel documento rispetto al numero di volte in cui il termine appare in tutti i documenti della raccolta.

$$score(q, d) = coord(q, d) * queryNorm(q, d) * \sum_{t \in q} (tf * idf^2 * boost(t) * norm(t, d))$$

$$tf(t) = (\#occur(t))^{1/2}$$

$$idf(t) = 1 + \log\left(\frac{numDocs}{docFreq + 1}\right)$$

**coord** fattore basato sul numero di termini della query trovati nel documento

**queryNorm** fattore di normalizzazione utilizzato per rendere comparabili gli score di query diverse

**tf** term frequency ( $tf_{i,j} = \frac{n_{i,j}}{|d_j|}$ ), dove  $n_{i,j}$  è il numero di occorrenze del termine  $i$  nel documento  $j$ , mentre  $|d_j|$  è la dimensione del documento  $j$  in numero di termini

**idf**<sup>2</sup> inverse document frequency ( $idf_i = \log \frac{|D|}{|\{d : i \in d\}|}$ ), dove  $|D|$  è il numero di documenti della collezione e il denominatore è il numero di documenti in cui è presente il termine  $i$

**boost** fattore di importanza dato a ciascun termine della query

**norm** racchiude fattori di importanza assegnati al documento e al campo, e un fattore di normalizzazione in base alla lunghezza del campo

Lucene aggiunge anche alcune funzionalità e perfezionamenti a questo modello, in modo da supportare la ricerca booleana e fuzzy, ma rimane essenzialmente un sistema basato su VSM.

---

<sup>2</sup>tf-idf, ottenuta come prodotto delle due misure tf e idf, è una funzione utilizzata nell'IR per misurare l'importanza di un termine rispetto ad un documento o ad una collezione di documenti

## 2.2 Agenti intelligenti e assistenti digitali

I chatbot hanno preso sempre più piede in ambito business (si parla oggi-giorno di *Bot Economy*) e se consideriamo che siano ancora, metaforicamente parlando, nella propria fase embrionale, ciò lascia ipotizzare un futuro in cui l'applicazione nelle varie funzioni aziendali, dalla customer care alla vendita e al marketing, possa diventare una costante. A conferma di quanto cita Gartner in una sua previsione<sup>3</sup>, l'**Intelligenza artificiale** sarà implementata ovunque, e il virtuale sarà sempre più una realtà concreta.

"By 2020, customers will manage 85% of their relationship with the enterprise without interacting with a human."

Intelligenza artificiale e robotica, come argomentato ampiamente in uno studio eseguito dal gruppo Unipol [10], hanno tutte le carte in regola per riuscire ad innovare il settore assicurativo, sia internamente che nella relazione col cliente, in un rapporto che si suole definire sinergico e di complementarità tra uomo e macchina. Queste due nuove tecnologie, citate tra le venti emergenti della Quarta Rivoluzione Industriale dal World Economic Forum<sup>4</sup>, renderanno i sistemi più all'avanguardia nel campo capaci di imparare dai dati, deducendone informazioni, e applicando quanto appreso nelle successive elaborazioni, in maniera del tutto autonoma.

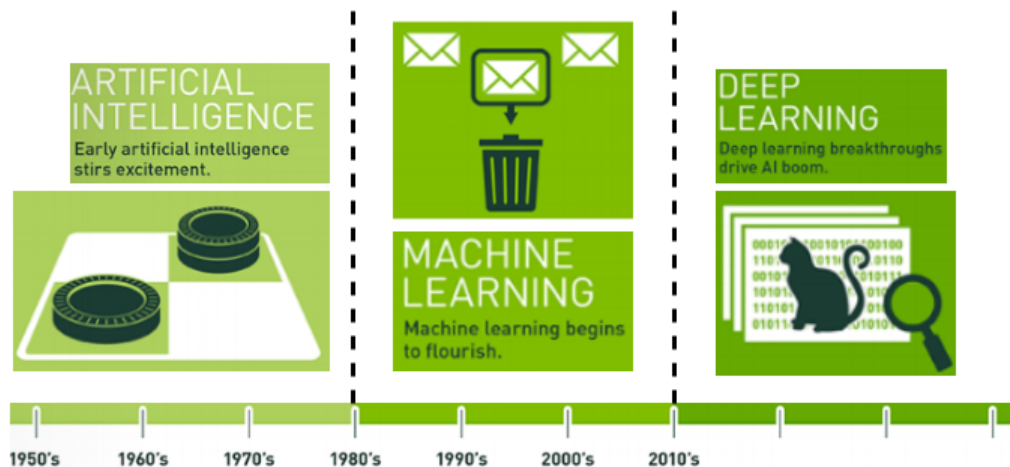


Figura 2.5: Machine Learning e Deep Learning al servizio dell'IA

Tramite il **Machine Learning** (apprendimento automatico), l'Intelligenza artificiale riuscirà ad essere realmente di supporto all'uomo, potenziandone o

<sup>3</sup>Gartner Summits, Gartner Customer 360 Summit 2011, March 30 - April 1, Los Angeles, CA

<sup>4</sup>WEF, The Global Risks Report 2017, 12th Edition, 2017.

integrandone le sue capacità. Di rilevanza, quindi, sarà il risultato raggiunto in termini di coesistenza tra le due intelligenze, quella umana e quella artificiale: i sistemi di IA non sono pensati per sovrastare ed eludere l'intelligenza umana, ma piuttosto perché la affianchino e si integrino/completino in maniera reciproca.

### 2.2.1 NLP: cosa si cela dietro ad un Chatbot

I chatbot sono dei sistemi automatici capaci di comprendere il significato del linguaggio umano, valutarlo e contestualizzarlo, al fine di garantire un dialogo rilevante.<sup>5</sup> In altri termini, un chatbot altro non è che un applicativo in grado di interagire con l'essere umano attraverso un sistema di messaggistica. Il chatbot è addestrato per tradurre i dati di input ricevuti in un valore di output desiderato. *L'elaborazione del linguaggio naturale* imita gli schemi del linguaggio umano in modo da simulare un tono umano nell'interazione uomo-macchina. In aggiunta, è possibile dotare il bot di una feature di analisi predittiva, per consentirne la generazione di informazioni in modo *proattivo* piuttosto che in risposta ad un prompt.

Un indicatore di come e quanto tali macchine possano essere considerate 'umane' è stato sviluppato intorno agli anni '50 dallo scienziato britannico Alan Turing. Il suo test verifica le capacità di pensiero e intelligenza in una macchina, e se è in grado di ingannare un essere umano facendogli credere che anche il bot stesso sia un umano, allora il *Test di Turing* è passato con successo. Sebbene inizialmente era dato per assodato che una macchina non potesse essere al livello di intelligenza dell'uomo, dal 1991, anno in cui ebbero inizio le competizioni Loebner Prize<sup>6</sup>, si iniziò a credere che ciò non fosse per niente scontato.

### 2.2.2 Classificazione degli agenti conversazionali

Per creare un chatbot, oggi, v'è un'incredibile varietà di tools e piattaforme, con differenti livelli di complessità, potere espressivo, e capacità di integrazione.

Allo stato attuale esistono tre classi principali di chatbots[11].

---

<sup>5</sup>Pat Group, infinite solutions, 'Trend Chatbot: tutto ciò che dovete sapere!', 2017

<sup>6</sup>si veda <http://www.aisb.org.uk/events/loebner-prize>

**Purposeless mimicry agents** La categoria più semplice è senza dubbio quella dei *purposeless mimicry agents*, che forniscono solo l'illusione della conversazione. Membri di questa classe sono **ELIZA** e i chatbots basati su dei modelli di ML **sequence-to-sequence**(StoS).

ELIZA (scritto da J.Weizenbaum, nel 1966) consisteva in poche e semplici *substitution rules*, che avevano lo scopo di imitare il ruolo di un terapeuta degli anni '60. Il chatterbot<sup>7</sup> ELIZA rispondeva al paziente rispondendo con altrettante domande ottenute dalla riformulazione delle affermazioni precedenti del paziente.



Figura 2.6: Esempio di conversazione con il chatbot Eliza

Fu chiamato così prendendo spunto da Eliza Doolittle, la protagonista di una commedia di G.Bernard Shaw che, grazie al metodo d'insegnamento della ripetizione delle corrette forme di pronuncia, apprende il raffinato modo di esprimersi delle classi sociali più agiate.

I moderni mimicry agents fanno uso del Deep Learning per imparare dalle conversazioni d'esempio. Il training set sul quale allenano la propria intelligenza è costituito da un insieme di dialoghi, e imparano a generare la prossima risposta nel dialogo a partire dall'ultima ricevuta dall'utente.

Il modello di DL utilizzato è solitamente un modello sequence-to-sequence[12], basato sulle *Recurrent Neural Networks* (RNN). Le reti neurali in oggetto (che possiedono una 'memoria' di ogni parola che passa attraverso di esse) possono essere usate alla stregua di modelli linguistici per predire elementi futuri

---

<sup>7</sup>Un chatterbot è un software progettato per simulare una conversazione intelligente con esseri umani tramite l'uso della voce o del testo, e vengono usati soprattutto per chiacchierare

di una sequenza, dati gli elementi precedenti della sequenza stessa. Un modello StoS aggiunge al modello linguistico i due step di codifica e decodifica. In una prima fase, *encoding*, il modello converte una sequenza di input (ad esempio una frase in lingua inglese) in una rappresentazione fissa (passata poi al decoder come una variabile di contesto). Nella fase di *decoding*, invece, un modello linguistico viene allenato sia sulla sequenza di output (frase tradotta) sia sulla sequenza fissa derivante dall'encoder.

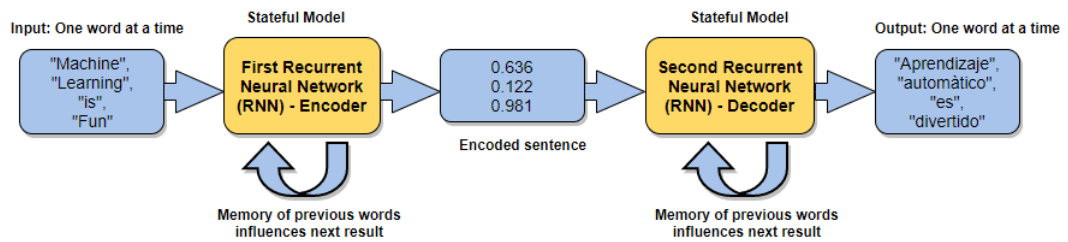


Figura 2.7: Esempio standard di applicazione di un modello StoS

L'applicazione di un modello StoS non è limitata alla mera traduzione del testo, ma può trovare riscontri positivi nei campi più disparati. Un paio d'anni fa circa, i ricercatori di Google[13] hanno dimostrato che è possibile servirsi di questi modelli per costruire dei Bot intelligenti. Il tutto ha avuto inizio con la raccolta di un log delle chat contenenti i messaggi scambiati tra dipendenti e il team di supporto tecnico di Google. Il modello prevedeva come frase d'ingresso la domanda del dipendente, mentre alla 'traduzione' associata nel modello veniva assegnata la risposta del team. Il risultato è stato un bot semi-intelligente in grado di rispondere (talvolta) alle domande di assistenza tecnologica. I ricercatori di Google hanno poi provato a costruire un bot a partire dai sottotitoli di film, con l'intento di 'insegnare' al bot a dialogare come un essere umano. Un altro team di ricercatori Google [14] ha rivisitato l'implementazione del modello sostituendo la prima RNN con una *Convolutional Neural Network* (CNN) e consentendo l'utilizzo di immagini, invece di frasi, come input al modello. Infine, altri ricercatori[15] hanno lavorato sul problema inverso, andando a generare un'intera immagine a partire da una descrizione di testo.

I problemi legati all'uso di modelli sequence-to-sequence sono essenzialmente due: anzitutto, essi non fanno uso di alcun meccanismo o conoscenza speciale per la comprensione della lingua, e inoltre risultano insensibili alle specifiche.

**Intention-based agents** I chatbot appartenenti alla classe degli *intention-based agents* sono in grado di comprendere dei comandi, e utilizzano questa loro abilità per eseguire dei compiti. Sotto tale classe ricadono, ad esempio, *Alexa* di Amazon, *Google Home*, *Cortana* di Microsoft, e *Siri* di Apple. Affinchè il chatbot sia capace di interpretare alla stregua di un comando ciò che l'utente richiede è necessario identificare il cosiddetto *intento* (ciò che l'utente vuole che l'agente conversazionale faccia), e capire i dettagli relativi all'intento identificato, di modo che l'azione possa essere compiuta.

L'assistente può determinare l'intento facendo uso di *keywords* o di *classificazione text-based*. Per l'utilizzo di parole chiave si procede definendo l'associazione di parole e/o frasi con gli intenti. Per una classificazione basata sul testo, invece, ciò che si fa è etichettare un set di frasi con gli intenti corretti e addestrare un classificatore ad-hoc. Se si dispone di una gran quantità di dati già etichettati, un'opzione valida è rappresentata dal servirsi del Deep Learning per mezzo di una CNN.

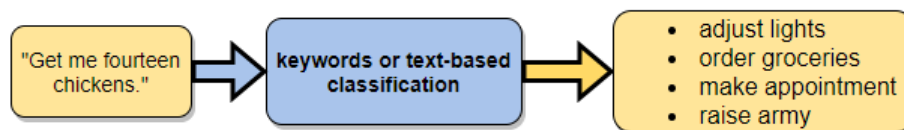


Figura 2.8: Intention-based agent: identificazione dell'intento

Una volta identificato l'*intent*, il prossimo step consiste nel convertire i dettagli relativi alla richiesta in un formato che sia comprensibile dalla macchina (un dizionario Python, ad esempio, come mostrato di seguito).

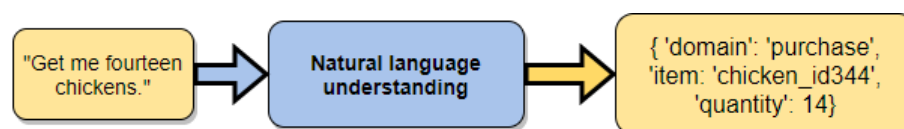


Figura 2.9: Intention-based agent: identificazione dei dettagli

Avere il comando disponibile nella forma di dizionario equivale ad adottare una semantica denominata *frame and slots*, dove il frame è rappresentato dal topic mentre gli slots sono le singole features e i valori corrispondenti. Reso machine-readable il modulo, il bot è in grado di fare ciò che deve, come se si trattasse di una comune istruzione.

**Conversational agents** La classe dei *conversational agents* si espande in un certo senso sulla classe precedentemente descritta, ma con la peculiarità della

conversazione a più turni. Per far sì che ciò sia possibile, è necessario che sia tenuta traccia dello stato corrente della conversazione, e quindi riuscire a capire quando l'utente intende parlare di qualcos'altro. Agenti conversazionali appartenenti a tale classe hanno quindi bisogno della presenza di un **dialog manager** per la gestione delle lunghe conversazioni, in maniera tale da essere in grado di accodare temporaneamente gli argomenti correnti e gestirne eventualmente di nuovi.

**RavenClaw** Il più noto gestore di finestre di dialogo è probabilmente *RavenClaw*<sup>8</sup>. RavenClaw è costituito da diversi agenti di dialogo, che non sono altro che dei piccoli applicativi organizzati gerarchicamente e che corrispondono a vari ambiti di conversazione. Le particolarità di RavenClaw sono rappresentate dalla presenza di un *dialog stack* e di una *expectation agenda*. Il primo è uno stack di agenti di dialogo che tiene traccia di tutte le possibili cose di cui il chatbot potrebbe voler parlare. La seconda, invece, è una struttura dati che tiene traccia di ciò che il chatbot si aspetta di sentire.

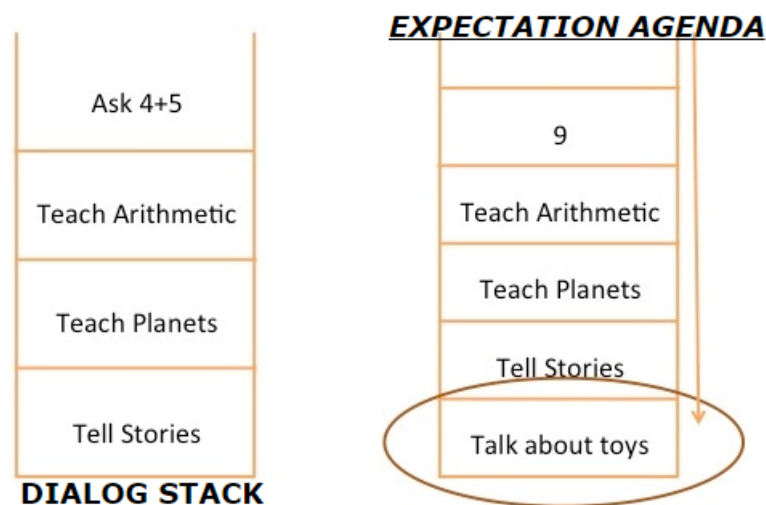


Figura 2.10: Dimostrazione di utilizzo del framework RavenClaw

Durante l'esecuzione del dialogo, il gestore si alterna tra le fasi di esecuzione e di input. La prima fase prevede la chiamata all'agente di dialogo superiore nello stack di dialogo, ed è lasciato parlare. Durante questa fase viene stabilito inoltre ciò che il bot si attende di ottenere come risposta, e registrato nell'agenda delle aspettative. Nella fase di input, infine, RavenClaw

<sup>8</sup>Dialog management framework sviluppato dalla Carnegie Mellon University e disponibile al download presso <http://www.cs.cmu.edu/~dbohus/ravenclaw-olympus/index-dan.html>



elabora ciò che l'utente dice e aggiorna la sua conoscenza, di modo che si riesca a determinare quale agente di dialogo può rispondere durante la fase di esecuzione.

Nell'esempio riportato in figura si evince che il chabot si attenda di ricevere '9' come risposta al problema matematico, ed è questa la ragione per la quale è la risposta in cima all'agenda delle aspettative. Se la risposta ottenuta, invece, è un'altra, l'algoritmo ricerca nell'agenda ciò di cui si potrebbe parlare, adeguandosi quindi all'input dell'utente (in tal caso, una domanda con tema ludico). Di conseguenza il chatbot sposterebbe la finestra di dialogo evidenziata in figura in cima al dialog stack, in modo da generare una risposta appropriata, e modificherebbe di conseguenza la sua agenda delle aspettative per far sì che sia nuovamente allineata a ciò che il chatbot si aspetta di sentire in risposta alla nuova domanda.

RavenClaw non è basato sul ML, ma in alternativa è possibile utilizzare un approccio basato sull'apprendimento. La predisposizione di un agente conversazionale può essere vista come un problema di **Reinforcement learning** (RL)<sup>9</sup>, all'interno di un processo decisionale di Markov (MDP) costituito da un insieme di *stati*, *azioni*, e una *reward function* che prevede una 'ricompensa' per essere in un dato stato  $s$  e svolgendo un'azione  $a$ . I due concetti di stati e azioni calati nel contesto di un chatbot assumono la seguente interpretazione. Uno stato non è altro che un insieme di cose che il chatbot conosce (ad esempio, le domande a cui ha già risposto), l'ultima cosa detta dall'utente o dal chatbot stesso. Le azioni vengono eseguite facendo particolari affermazioni, e la ricompensa deriva dal raggiungimento di un obiettivo (ad esempio, l'utente che completa con successo la prenotazione di un viaggio).

Due problemi di fondo vengono fuori a questo punto. Sebbene l'apprendimento per rinforzo possa essere utilizzato al fine di imparare una policy che dia la migliore azione da compiere in un certo stato, è chiaro che per ottenere un simile risultato sia necessario un cospicuo addestramento del modello. Inoltre, non ci si dimentichi del fatto che risulta difficile sapere con esattezza in che stato l'agente si trovi in un dato istante, a causa di errori nel parlato come anche errori di comprensione. Ecco spiegato il motivo per cui questo tipo di agenti siano ancora in fase di sperimentazione.

Lo stato dell'arte attuale è racchiuso nel seguente assunto: risulta possibile, ed è la strada che è attualmente percorsa, dotare un chatbot di sufficiente conoscenza per far sì che questo sia in grado di cooperare con l'umano sullo svolgimento di tasks che richiedono una comprensione minima ma, per

---

<sup>9</sup>'Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning', 2002, K.Scheffler, S.Young

costruire dei chatbot migliori, si necessita di ricerca e di passi avanti nell'ingegneria della conoscenza, cosicché gli agenti conversazionali si adattino e meglio seguano le sottigliezze di significato e legate alla conversazione.

### 2.2.3 Bot economy: ecco come i chatbots daranno una svolta all'economia

La domanda che ci si pone è se il mercato dei chatbot possa o meno riuscire a creare ricchezza a tal punto da essere considerato un'economia. I dati resi pubblici da 'Transparency marketing research' nel dicembre 2016[16] dimostrano che il fatturato generato dalla Bot Economy è destinato ad un incremento continuo e decisamente sostenuto, passando su scala mondiale dai 113 milioni di dollari (2015) ai 994.5 milioni (2024). Il fenomeno non è isolato ma si inserisce nel trend di crescita esponenziale del fatturato legato all'utilizzo dell'intelligenza artificiale.

Oltre ad aver creato un nuovo mercato di scambio, la Bot Economy ha prepotentemente modificato le relazioni tra consumatori ed imprese. Il mercato dei chatbot ha visto dunque crescere a dismisura la propria portata e la propria notorietà, e di recente la tecnologia ha iniziato a dimostrare di poter arrecare reale beneficio alle aziende.

**Benefici sul settore assicurativo** Sono molteplici le applicazioni dell'IA di supporto alle attività umane e, nondimeno, quelle in un contesto quale quello assicurativo o, più in generale, in un contesto ove l'interazione col cliente assume primaria importanza (vedi servizi in grado di rispondere in modo sempre più puntuale alle esigenze del singolo cliente). L'impatto nell'utilizzo sempre più consueto di queste tecnologie, sia internamente che esternamente al mondo assicurativo, non sarà di poco conto.

Sul fronte interno, la totalità dei processi subirà presumibilmente delle attività di snellimento e ottimizzazione, riuscendo in questo modo a ridurre sensibilmente i costi di gestione ed incrementare le potenzialità di business. Attività ripetitive e/o a basso valore aggiunto per la Compagnia Assicurativa saranno affidate alle nuove tecnologie, guadagnandone in termini di efficienza ma anche di conformità alle richieste normative (verificabilità, sicurezza, qualità dei dati, resilienza operativa). È da sottolineare, infine, che le Compagnie Assicuratrici dispongono di un gran patrimonio di dati e, nell'ottica del Machine Learning, non si dimentichi il ruolo importante che rivestono i dati come fonte di alimentazione dell'apprendimento automatico. Ne segue che l'IA sarà d'aiuto alla Compagnia nell'analisi dei clienti, andando a definire in maniera

esaustiva ma al contempo rapida quelli che sono i loro bisogni.

Per quanto riguarda, invece, l'impatto esterno al mondo assicurativo, è chiaro che gli effetti prodotti dall'utilizzo sempre più insistente di queste tecnologie avranno delle ripercussioni nel quotidiano delle aziende e dei singoli individui. Nuove forme di copertura e servizi innovativi dovranno rappresentare la pronta risposta dell'ente assicurativo, nell'ottica di un adattamento rapido e continuo all'evoluzione del trend.

**Benefici sul settore finanziario e della Customer care** Sempre più Istituzioni Finanziarie investono e utilizzano questi strumenti, creando dei prodotti sempre più in grado di profilare l'utente e andare il più possibile incontro alle loro reali necessità. È evidente che più un chatbot è in grado di imparare dalle abitudini del consumatore, più diventa facile offrire dei prodotti su misura per il cliente.

L'evoluzione continua del settore dell'Assistenza clienti testimonia il fatto che questo sia uno tra i settori che più beneficiano di questa tecnologia. Tale tecnologia, oltre che migliorare il servizio clienti, può trasformarlo facendo in modo che diventi proattivo nei confronti del cliente, e dunque non di semplice assistenza.

# Capitolo 3

## Descrizione della soluzione

*Le banche dati archiviano generalmente le loro informazioni ordinatamente e organizzate secondo una struttura rigida che, il più delle volte, risulta più che sufficiente per consentirne la pura e semplice ricerca per campi tipica di un DBMS. Ma non sempre è così. La ricerca in un database strutturato non è orientata alla ricerca del testo e al relevance ranking. Ecco che, sovente, i motori di ricerca rappresentano la soluzione giusta anche in un contesto quale una base di dati.*

*Nel presente capitolo verrà dapprima fornita una visione d'insieme della soluzione oggetto di tesi e poi presentata nel dettaglio l'architettura sottostante. Seguirà un'analisi del sistema e delle componenti utilizzate.*

*La presentazione delle scelte implementative e i dettagli legati alla messa in atto del sistema saranno poi discussi nel capitolo IV.*

### 3.1 Presentazione generale dell'architettura

Il sistema si compone essenzialmente dei tre macroblocchi relativi a:

1. recupero dei dati da un modello relazionale relativo ad un ambito assicurativo
2. indicizzazione dei dati raccolti, previa elaborazione di questi in un formato *view-document*
3. ricerca e presentazione dei risultati all'utente finale, per tramite di una *web application* integrata con un chatbot di supporto all'utente

L'architettura presentata prevede che vi sia una fonte alimentante dei dati, che coincide esattamente con un database contenente tutte e sole le informazioni relative ad un cliente e gestite da una Compagnia assicurativa. Il contesto applicativo prende il nome di CRM, acronimo di *Customer Relationship*

*Management* (gestione delle relazioni con i clienti), ovvero una soluzione software adottata da una Compagnia al fine di organizzare un database di informazioni (patrimonio informativo) indispensabile per gestire efficacemente il rapporto con i propri clienti. Le applicazioni CRM servono essenzialmente a tenersi in contatto con la clientela, a inserire le loro informazioni nel database e a fornire loro modalità per interagire, in modo che tali interazioni possano essere registrate e analizzate. È questa l'ottica in cui si inserisce il lavoro di tesi presentato.

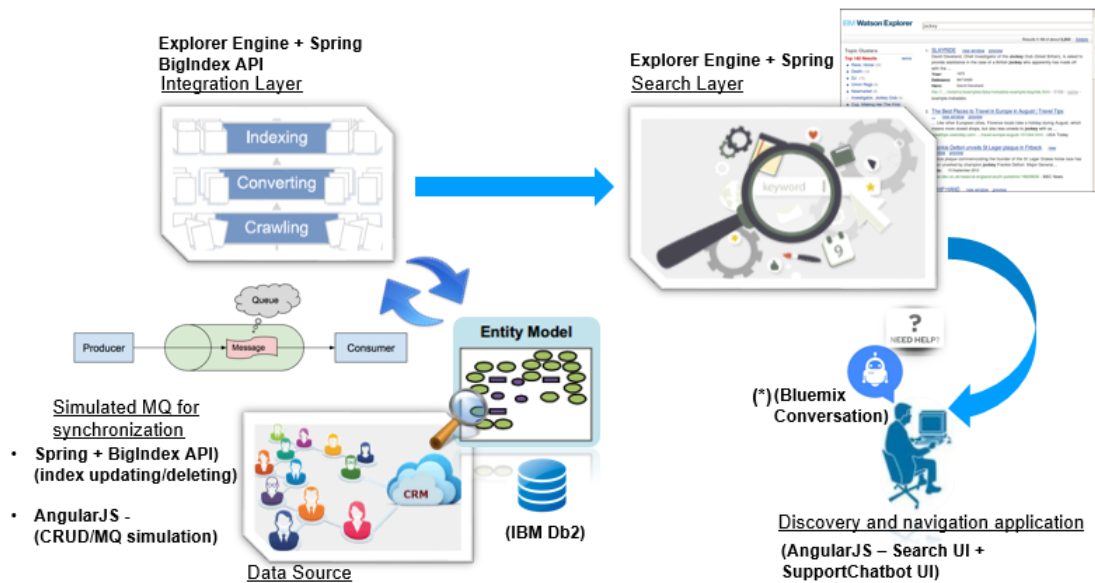


Figura 3.1: Architettura del sistema

In un database i dati sono strutturati in modo tale da consentire la gestione degli stessi in termini di inserimento, aggiornamento, cancellazione e ricerca delle informazioni. Nel dettaglio, per migliorare i tempi di ricerca dei dati, in un db viene adottata una struttura dati che è quella dell'*indice*, che viene generato in automatico dal sistema per i campi di una tabella definiti come chiave e manualmente per i campi per cui è necessario effettuare delle operazioni di ricerca o di join. Questo perché, in assenza di indice, la ricerca del valore di un campo viene eseguita in maniera sequenziale sui record della tabella. Ciò descritto consente a tutti gli effetti di perseguire il fine ultimo rappresentato dal *Information Retrieval*, ovvero il soddisfacimento del *bisogno informativo* dell'utente. A titolo di esempio, infatti, si ipotizzi la necessità di portare a termine il task di reperimento, in un database aziendale, di tutti e soli gli impiegati il cui nome sia 'Rossi'. La query SQL di ricerca 'Select \* from impiegati where Cognome like 'Rossi' sarebbe più che sufficiente a raggiungere il completamento del task, e tra l'altro con valori di precisione e recall

molto alti. Ciò, però, mal si adatta al perseguimento di una soluzione che vada al di là della mera interrogazione di una base di dati. I dati strutturati sono caratterizzati da una struttura rigida quale quella tabellare, e le query di ricerca ereditano essenzialmente i limiti dell'algebra booleana, anche e soprattutto in termini di capacità di ricerca e di recupero dei risultati.

Quindi, identificati e raccolti i dati messi a disposizione dalla Compagnia, ci si serve degli strumenti offerti dalla Suite **Watson Explorer Foundational Components** per costruire l'insieme dei documenti associati al cliente, per una data agenzia, e avvalersi della programmatica creazione, distribuzione e gestione degli indici creati (**Integration Layer**), e predisporre tali documenti all'interrogazione da parte del **Search Layer**, che si occupa di prendere una query immessa dall'utente ed opportunamente elaborarla al fine di restituire l'insieme dei risultati della ricerca.

L'indice costruito viene mantenuto consistente con il dato originario grazie ad un meccanismo simulato di *coda di eventi* (MQ), in base al quale una modifica sulla banca dati viene riversata sulla collezione dei documenti di modo che l'indice interessato dall'evento sia coerentemente ricostruito.

La piattaforma di ricerca assistita offre all'utenza la possibilità di soddisfare il proprio bisogno informativo per mezzo di due differenti modalità di ricerca esposte dal sistema. Il tutto è reso trasparente all'utente che, previa compilazione di un semplice form di ricerca, si limita ad esprimere la sua richiesta, che verrà elaborata da un motore di ricerca *Google-like*.

Per far fronte, infine, alla necessità dell'utente di avere un'interazione rapida e facilitata con il portale di ricerca, quest'ultimo integra al suo interno un *Customer Support chatbot*, che vestirà i panni di un assistente digitale in grado di fornire guida e supporto all'utente nella sua attività di ricerca.

## 3.2 Fonte dei dati trattati

Il modello relazionale dei dati caratterizzante il database assicurativo presenta un'entità primaria che è quella del cliente (*Cliente*), ed una serie di entità, definibili secondarie, correlate al cliente stesso e che descrivono le informazioni derivanti dal legame tra il cliente e la Compagnia (*Portafoglio*<sup>1</sup>, *Indirizzo*, *Recapiti*, *Veicoli*, e *Sinistri*). La soluzione presentata limita il campo d'interesse alle entità Cliente, Portafoglio e Indirizzo, ma nulla vieta di considerare l'insieme completo delle entità, considerata la facile e rapida estensibilità della soluzione.

---

<sup>1</sup>Insieme delle polizze emesse da una compagnia assicurativa in una determinata data

L'entità Cliente raccoglie i dati relativi all'anagrafica (nome, cognome, data di nascita, etc.) di un cliente, ovvero l'informazione condivisa tra i diversi segmenti di Agenzia e quindi indipendente dalla particolare Compagnia-Agenzia. La relazione (molti-a-uno) definita da ciascuna delle entità secondarie verso l'entità primaria del Cliente è percorribile mediante la chiave esterna CONT\_ID (identificativo del Cliente).

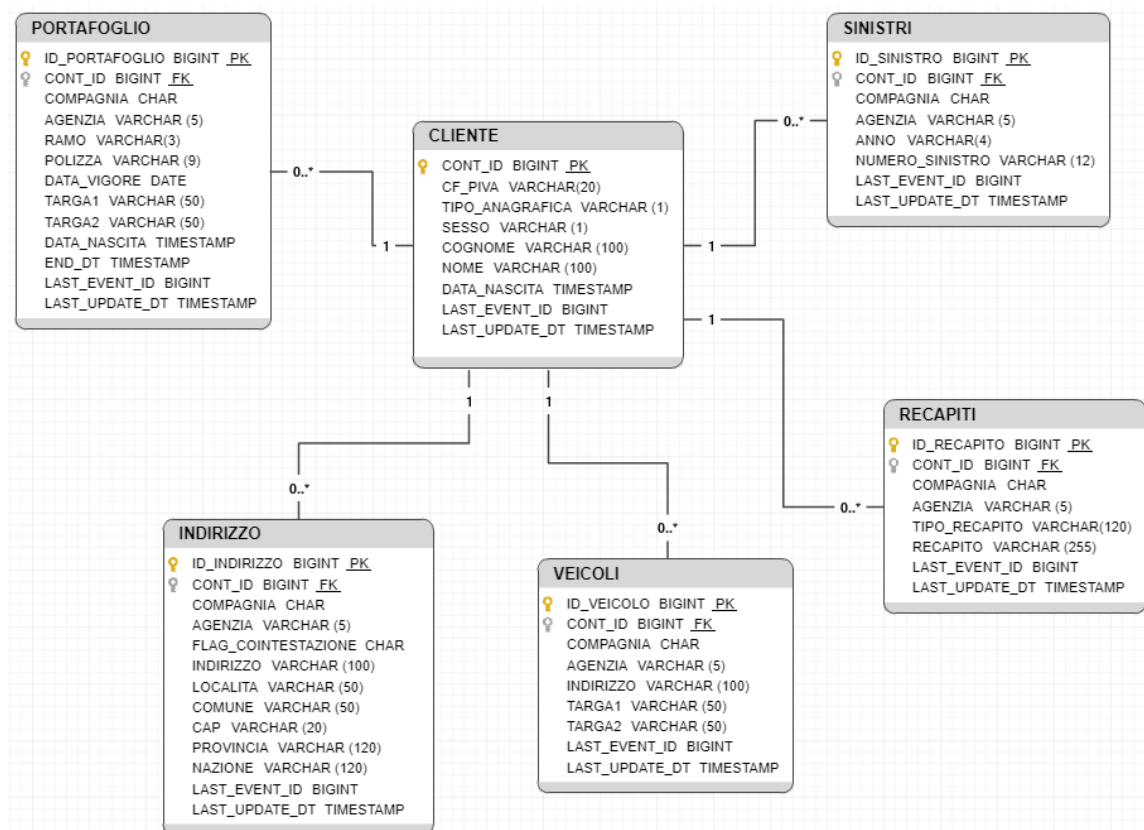


Figura 3.2: Modello relazionale dei dati

Per un dato cliente, un particolare segmento di agenzia (specificato dalla coppia Compagnia-Agenzia) conserva le informazioni legate all'indirizzo e ai portafogli del cliente registrati presso quel particolare segmento.

**Strumenti utilizzati** Per la gestione della fonte dati viene utilizzato **IBM® Db2®**, un Relational Database Management System (RDBMS) proprietario di IBM. Db2 offre prestazioni, flessibilità, scalabilità e affidabilità, rappresentando una ottima scelta per le soluzioni aziendali. Può essere usato all'interno di software applicativo accedendo a una serie di APIs disponibili per numerosi linguaggi di programmazione, tra cui Java (il linguaggio scelto nel lavoro di

tesi per l'implementazione di Integration Layer e Search Layer) e altri quali Python, Ruby, Perl, PHP, C, C++, COBOL, Fortran, o altri linguaggi che supportano il Framework .NET.

Per interfacciarsi con il database e poter usufruire delle operazioni di CRUD (Create-Read-Update-Delete) viene invece fatto uso di:

**SQuirreL SQL Client** uno strumento di amministrazione di database, gratuito e open source, scritto interamente in Java e distribuito tramite licenza GNU.

SQuirreL è un client che usa driver JDBC per consentire all'utente di esplorare e interagire con un DBMS di diverso tipo attraverso un'interfaccia utente

**Una custom web application AngularJS** un'interfaccia web realizzata servendosi del framework JS, per facilitare e migliorare l'interazione standard con la fonte dei dati, nonché consentire la riproduzione del meccanismo utilizzato per mantenere consistenti gli indici

### 3.3 Indicizzazione e ricerca dei documenti

**Integration Layer** Identificata la sorgente dei dati, il primo step è affidato al macroblocco dell'indicizzazione. L'applicativo costruito gestisce una prima operazione di **Initial Load** dei documenti e l'attività di **rigenerazione** dell'indice associato ad un certo documento al verificarsi di una condizione quale un inserimento, una modifica, o una cancellazione di un record d'interesse nella base di dati. Nel dettaglio, all'atto del verificarsi di una delle tre condizioni sopra citate, viene recuperato (qualora disponibile, perché in caso di cancellazione di un record, questo non figura più nel database) il particolare record e viene ricostruito il documento del cliente relativo al *segmento di agenzia* identificato dal record stesso.

Il documento generato in fase di indicizzazione rappresenta una vista del cliente che racchiude al suo interno sia le informazioni di carattere generale legate all'anagrafica sia quelle legate alla particolare Compagnia-Agenzia per la quale la vista del cliente è stata generata. Ciò che ne consegue è che, per un cliente, la collezione indicizzata conterrà al suo interno un insieme di documenti-vista associati ad esso, uno per ogni segmento di agenzia presso il quale il cliente è registrato. In aggiunta, viene costruito e indicizzato un documento rappresentativo dei soli suoi dati anagrafici, di modo da avere, per ogni cliente, un documento ricercabile a livello di *gruppo*, ossia un documento



che racchiude solo ed esclusivamente le informazioni condivise sul cliente e non riporti alcun dettaglio su informazioni quali indirizzo o portafogli.



Figura 3.3: Document-view relativa ad un cliente

**Search Layer** Indicizzati i documenti e predisposti, quindi, gli stessi all'attività di ricerca, il macroblocco successivo coincide con la costruzione di un layer che, essenzialmente, si occupa di gestire le richieste che l'utente ha modo di effettuare dal portale di ricerca (opportunamente mappate e tradotte in forma di query dalla logica presente nella piattaforma, e sottoposte al motore di ricerca). L'applicativo ideato, dunque, ha il compito di richiamare le API esposte dal search engine Watson e fornire l'insieme dei risultati, che saranno poi visualizzabili dall'utente nella web application. Le potenzialità delle quali si è dotata la piattaforma non si limitano alla ricerca che un tradizionale database riuscirebbe a riprodurre, ma includono un insieme di funzionalità che ampliano e arricchiscono la modalità di ricerca fruibile dall'utente:

- ricercare tra le informazioni condivise a livello di *gruppo* o tra quelle specifiche del *segmento di agenzia*
- far insistere la ricerca solo ed esclusivamente su particolari concetti (nome, indirizzo, polizza, etc.)
- filtrare la ricerca in base a specifici criteri, *restringendo* il campo dei documenti ricercabili
- ricercare, secondo una logica corrispondente all'operazione booleana di *OR*, i termini specificati nella richiesta

- abilitare, selettivamente (sulla richiesta in toto o su di un subset preciso di termini) la ricerca del contenuto per *wildcard matching* (ricerca parziale)
- in alternativa ai tradizionali criteri di *ordinamento* per valore di un concetto, è previsto un ordinamento per *ranking*, basato su criteri complessi mutuati dai web search engines, che cercano di produrre risultati più vicini possibile all'ipotizzata massima rilevanza per l'utente

**Strumenti utilizzati** IBM Watson Explorer Foundational Components[17] è una suite di strumenti a supporto dello sviluppo di soluzioni *360-degree* di *discovery* ed *enterprise search*. Le funzionalità offerte dalla suite sono presenti in entrambe le edizioni di Watson Explorer ad oggi disponibili sul mercato (Enterprise e Advanced edition).

La versione di IBM Watson Explorer utilizzata nel lavoro di tesi è la 11.0. Storicamente l'edizione di Watson Explorer (WEX) come è conosciuta oggi-giorno nasce dal prodotto precedente 'IBM Content Analytics with Enterprise Search' (ICAwES), unito alle funzionalità del prodotto acquisito 'Vivisimo'. L'acquisizione di Vivisimo ha portato alla nascita dei prodotti Infosphere Data Explorer e Watson Explorer, prima di arrivare alla versione finale di *Watson Explorer Enterprise* o *Advanced Edition*. Le due edizioni differiscono per quanto segue:

**Enterprise Edition** Consente l'accesso alle informazioni aziendali e le funzionalità delle applicazioni 360-degree in fonti e servizi cloud interni, esterni e ibridi, nonché la capacità di integrare i servizi cognitivi IBM Watson Developer Cloud (IBM Bluemix)

**Advanced Edition** Fornisce tutte le stesse funzionalità di Watson Explorer Enterprise Edition, nonché le funzionalità avanzate di analisi dei contenuti per consentire alle organizzazioni di adattare la propria soluzione di accesso alle informazioni a domini specifici, e di estrarre le informazioni da fonti di dati non strutturate, per identificare le tendenze, i pattern e i rapporti nei loro dati

Watson Explorer indicizza grandi volumi di dati strutturati, non-strutturati o semi-strutturati, dalle più svariate sorgenti di dati. In aggiunta, fornisce la capacità di costruire applicazioni di esplorazione dei Big Data e consente ai clienti di creare una vista virtuale delle informazioni rilevanti relative alle differenti entità coinvolte, ovvero clienti, prodotti, eventi e business partners.

Queste view sono costruite a partire da grandi set di dati memorizzati in differenti repository (interne o esterne) di dati, incluse CRM, *Enterprise Resource Planning* (ERP), *Content Management System* (CMS), database, e applicazioni di *Knowledge Management* (KM), il tutto potendo disporre di modelli di sicurezza e senza la necessità di 'spostare' i dati. Le organizzazioni con strategie di Big Data differenziate si troveranno in una posizione migliore rispetto alle dirette concorrenti, riuscendo a sfruttare appieno l'espansione del mercato e le opportunità emergenti, tutte abilitate e supportate da Watson Explorer.

La figura che segue fornisce una panoramica gerarchica dell'insieme delle funzionalità offerte dalle varie componenti di WEX e dal suo framework sottostante. Watson Explorer è costituito da due moduli:

**Engine** Watson Explorer Engine si contraddistingue per la sua capacità di *securely crawl* e *indicizzazione* di grandi volumi di dati

**Application Builder** Un'evoluzione del modulo Watson Explorer Engine, che si rivolge alle esigenze del crescente mercato dei Big Data. Application Builder (AB) consente alle organizzazioni di creare rapidamente e facilmente applicazioni *browser-based* che permettono l'analisi dei vari problemi di business nelle aree della *customer experience*, dell'ottimizzazione delle vendite, della ricerca e dello sviluppo. AB si serve dei dati orchestrati dal modulo Engine per creare modelli specifici del dominio, organizzare i dati relativi alle entità e creare/modificare opportuni widgets per soddisfare le esigenze aziendali. Application Builder aggiunge, infine, la possibilità di mostrare dati rilevanti e personalizzati in base alle relazioni di questi con altri dati rilevanti.

Oltre a queste applicazioni *stand-alone*, Watson Explorer include un valido set di API 'enterprise-ready', conosciuto con il nome di **BigIndex API**. Tali API forniscono interfacce ad alto livello per le funzionalità base dell'Engine e dell'Application Builder, come creare *search collections*, ricercare dei contenuti indicizzati, e restituire set organizzati dei risultati di ricerca.

### 3.3.1 Watson Explorer Engine

Watson Explorer Engine[18] offre un framework flessibile basato sull'utilizzo di specifici plug-ins *JVM-based*<sup>2</sup>, al fine di rendere possibile alle organizzazioni il recupero del contenuto da qualsiasi repository di dati accessibile dalla

---

<sup>2</sup>I *Connector plug-ins JVM*(Java Virtual Machine)-based sono scritti principalmente nei linguaggi di programmazione Java e Scala.

rete, dai file-systems alle specifiche applicazioni. WEX include, inoltre, meccanismi sofisticati per la standardizzazione, indicizzazione e successiva esplorazione dei dati.

L'architettura di Watson Explorer Engine è flessibile e modulare. La creazione degli indici è percorribile mediante:

1. *crawling* di una repository di dati, ovvero il recupero, da quest'ultima, dei dati, metadati e informazioni di sicurezza; quando l'attività di crawling risulta impossibile o inefficiente, WEX prevede il supporto ad applicazioni server-side che eseguono il *push* dei dati da una repository verso il modulo di Watson Explorer
2. *conversione* dei dati recuperati in un formato comune che è adatto alla successiva elaborazione degli stessi, nonché arricchimento dei dati recuperati con *meta-information* rilevanti, come l'URL (Universal Resource Locator) associata, l'originale *file type*, informazioni di autorizzazione, e un identificatore univoco per ciascun oggetto recuperato
3. creazione di un indice (*search collection* è il termine usato per descrivere una o più 'information sources', contenenti l'indice costruito di pari passo al crawling di tali sorgenti)

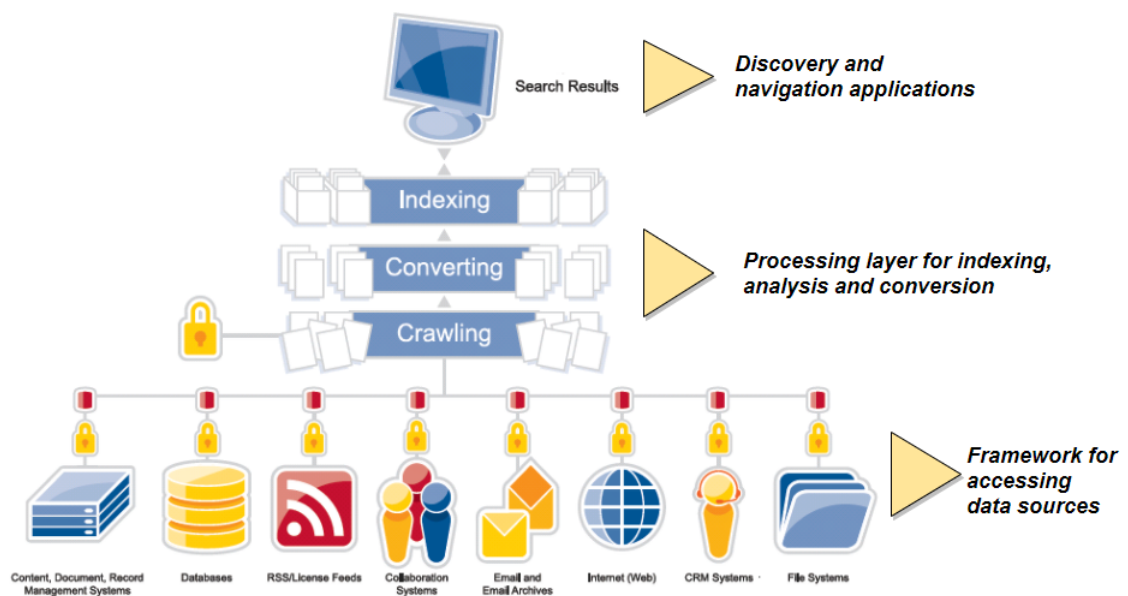


Figura 3.4: Architettura IBM Watson Explorer

**Connettori e crawling** Watson Explorer può recuperare ed indicizzare tutti i dati a cui è possibile accedere tramite un URL. Vengono forniti dei connettori

che consentono l'accesso alle repository remote per il recupero dei contenuti. Ad esempio, i connettori per i vari database utilizzano le *Java Database Connectivity* (JDBC) API per recuperare gli URL associati ai record del database stesso.

Una volta recuperati gli URL, questi sono accodati ad un componente di Watson Explorer Engine noto come *Crawler*, o rielaborati per facilitare il successivo accodamento. Il crawler, poi, recupera i dati e i relativi metadati associati a ciascun URL accodato, preparando questi alla successiva elaborazione eseguita dalle altre componenti. Dal momento che tali connettori recuperano sia gli URL che i dati dalle repository remote, ci si riferisce spesso a questi con il nome di *pull connectors*.

Un'alternativa al crawling delle repository remote è installare, lato-server, delle componenti capaci di accedere a queste repository, e che accodino direttamente i dati per l'attività di indicizzazione.

**Conversione dei dati per l'indicizzazione** L'attività di crawling è seguita dalla normalizzazione e, se necessario, arricchimento dei dati. I dati recuperati dal crawler sono tipicamente in un formato che è specifico per la repository (o relativo al tipo di file, se il dato è recuperato da una risorsa eterogenea, come un file system o un CMS).

Indicizzare tali dati caratterizzati da differenti formati equivarrebbe a far sì che l'indexer comprenda tutti questi formati, il che si tradurrebbe in un aumento della complessità, distogliendo l'attenzione da quello che è il fine ultimo di indicizzazione. Senza dimenticare che ciò richiederebbe anche opportune modifiche da apportare all'indexer ogni qualvolta Watson Explorer Engine necessiti di indicizzare una tipologia di documento precedentemente mai vista. Ciò che ne consegue è che per indicizzare rapidamente e accuratamente, e per mantenere il focus dell'attenzione sull'attività di indicizzazione piuttosto che su di un'attività di trasformazione dei dati, è prevista una normalizzazione di questi ultimi in un formato comune utilizzato internamente da WEX. Tale processo viene eseguito elaborando i dati tramite una sequenza di convertitori chiamata *pipeline di conversione*.

Ciascun convertitore presente nella pipeline converte i dati da un formato ad un altro, o li arricchisce estraendo e aggiungendo ulteriori informazione, prima di passare i dati al convertitore successivo nella sequenza. Il formato conclusivo di output è noto come *VXML* (Vivisimo XML, per ragioni storiche).

**Indicizzazione** Al fine di facilitare l'esplorazione dei dati e la navigazione tra le informazioni, Watson Explorer prevede l'utilizzo degli indici. I benefici

derivanti dall'indicizzazione sono essenzialmente:

- minimizzazione del traffico di rete
- simultaneità e parallelismo nell'attività di ricerca di più repository (l'alternativa sarebbe ricercare in ciascuna delle repository individualmente e sequenzialmente)

Sebbene i task di creazione dell'indice differiscano al variare della tipologia di repository, le modalità di esplorazione e di utilizzo di un indice sono comuni a qualsiasi applicazione Watson Explorer.

Al fine di supportare i requisiti RAS (*reliability, availability, and serviceability*) delle applicazioni enterprise, gli indici in WEX possono anche essere configurati in modo da essere replicati su istanze multiple di Watson Explorer Engine. Ciò è generalmente indicato come *distributed indexing*. Aggiornamenti in queste repliche, basati sulla modifica del contenuto nella repository dei dati associata, possono essere avviati da una qualsiasi replica, e sono automaticamente distribuiti a tutte le altre repliche. La sincronizzazione dell'indice viene eseguita a livello di transazione, per far sì che l'aggiornamento abbia successo e sia consistente tra tutte le repliche dell'indice.

**API Watson Explorer Engine** Per facilitare lo sviluppo, gli sviluppatori sono incoraggiati a configurare il maggior numero di oggetti possibile attraverso lo strumento amministrativo. Questo strumento fornisce un'interfaccia web-based e user-friendly per la creazione delle strutture XML utilizzate dalle applicazioni di ricerca di Watson Explorer Engine.

L'alternativa è rappresentata dal servirsi, nello sviluppo delle applicazioni di ricerca in WEX (siano queste stand-alone o embedded) di un IDE o un editor di testo ed utilizzare l'insieme di API di Watson Explorer Engine, tramite SOAP (Simple Object Access Protocol) o REST (Representational State Transfer).

C'è un grande dibattito sul Web su quale di questi due protocolli sia più facile, migliore o più appropriato da utilizzare per lo sviluppo. Sebbene SOAP fosse il protocollo più comune utilizzato nei primi tempi dei Web Services, REST è diventato il modo più popolare di offrire Web Services sul Web aperto. La ragione principale di questa tendenza è che REST è molto più facile da implementare: digitare un URL nel tuo browser, modificare alcuni parametri CGI ed è possibile far funzionare il servizio in poco tempo. Questa facilità d'uso è fondamentale per il mondo veloce del moderno sviluppo Web. I sostenitori di REST ne lodano dunque la leggerezza, la trasparenza e la facilità d'uso. Riassumendo, REST è comunemente usato quando:

- si desidera sviluppare un servizio in fretta ed averlo operativo prima possibile
- si prevede di utilizzare l'API per interazioni semplici, come per la ricerca
- si desidera sviluppare un servizio in fretta ed averlo operativo prima possibile
- risulta comodo manipolare XML e/o non si desidera utilizzare ambienti di programmazione pesanti come Visual Studio o Eclipse

I sostenitori di SOAP sostengono, invece, che REST non sia neanche da considerarsi un protocollo, e che SOAP è più deterministico e robusto (grazie al meccanismo di controllo dei tipi), e che è in realtà più facile da usare per via del supporto SOAP che è incorporato in molti strumenti di sviluppo.

L'API Watson Explorer Engine supporta entrambi i protocolli SOAP e REST con funzionalità identiche per semplificare l'integrazione di Watson Explorer Engine nella più ampia gamma possibile di applicazioni Web. Il suo design garantisce essenzialmente che l'API supporterà sempre sia SOAP che REST con lo stesso livello di funzionalità.

SOAP e REST presentano diversi vantaggi che giustificano l'utilizzo dell'uno o dell'altro in determinati scenari. A parità di condizioni, gli sviluppatori di applicazioni WEX spesso preferiscono SOAP semplicemente perché le applicazioni SOAP richiedono un'attenzione significativa ai dettagli che semplifica lo sviluppo, il debug e il mantenimento di applicazioni complesse.

### 3.3.2 BigIndex API

Watson Explorer BigIndex (BI)[19] fornisce un esaustivo set di API di livello enterprise per la programmatica creazione, distribuzione e gestione degli indici. Le API offerte nascondono la complessità che può nascere nell'approccio alla creazione e interrogazione degli indici, nonché nel ritornare i risultati di query eseguite su tali indici. BigIndex nasconde anche la complessità che si cela dietro la gestione delle repliche degli indici in Watson Explorer, offrendo le funzionalità di automatizzazione del *load* e del *balancing* su più server, quella di *fault-tolerance*, e quella legata alla possibilità di *migrazione* degli indici da un server ad un altro.

Le applicazioni che richiedono tali funzionalità possono sfruttarle e configurarle in maniera semplice tramite BigIndex API, eliminando le innumerevoli ore altrimenti necessarie allo sviluppo e al testing di queste per ciascuna applicazione. BI riduce dunque la complessità del codice e rende l'interfacciarsi

con un complesso sistema distribuito facile come lavorare con un client locale. Per poter garantire il soddisfacimento dei requisiti tipici di una applicazione enterprise, quali *high availability* e *load balancing*, gli indici creati con BigIndex sono in genere partizionati in un insieme di segmenti che prendono il nome di *shard*, che possono essere replicati sulle istanze di WEX associate con un dato indice. Segmentando gli indici in *shard*, mettendo a punto le dimensioni relative al generico *shard* e distribuendoli equamente attraverso i server, si riescono così ad ottenere dei benefici in termini di requisiti di memoria richiesti.

I server associati ad un indice sono identificati in una *cluster-collection-store* nel modello definito per un'entità dell'applicazione. La definizione dell'entità include anche attributi che specificano il numero di *shard* da associare all'indice e come l'indice viene utilizzato.

Le applicazioni create con BigIndex API consentono di definire e modellare una particolare entità sia programmaticamente (via codice) che mediante un approccio *configuration-driven*. L'approccio programmatico è più semplice in termini di setup richiesto per la configurazione, ma richiede che tutte le entità e i 'depositi' dei dati siano esplicitamente configurati per tramite delle API. Diversamente, l'approccio *configuration-driven* prevede la disposizione di opportuni file (XML-based) esterni di configurazione. Questo è sicuramente l'approccio più appropriato per un ambiente di produzione.

L'indicizzazione e la ricerca sono argomenti strettamente correlati: è possibile cercare i dati usando solo la struttura e gli attributi con cui li si indicizza. Quando viene creato un indice, quindi, deve essere considerato ciò che si cercherà e come. Tali domande aiutano lo sviluppatore a determinare i tipi di record da creare, il numero e i tipi di campi che tali record contengono, le proprietà dei campi, e come i record sono collegati tramite associazioni (e altro ancora).

Quando si indicizzano i dati, non si spostano fisicamente i dati stessi nell'indice; i dati rimangono nella posizione corrente e l'indice contiene informazioni sui dati. La dimensione dell'indice, tuttavia, può variare notevolmente a seconda dei parametri scelti durante l'operazione di indicizzazione. Si necessita di decidere accuratamente quali parti dei dati sono da indicizzare e la quantità di dati originali da includere nell'indice, effettuando tali scelte usando l'API BigIndex o facendo affidamento sul framework di conversione integrato e altamente configurabile di Watson Explorer Engine. Una volta generato un indice, è possibile fare affidamento sulle API BigIndex per mantenerlo aggiornato rispetto ai dati di origine. È possibile aggiornare un indice esistente aggiungendo nuove informazioni, eliminando le informazioni esistenti, o



modificando informazioni esistenti inviando aggiornamenti per riflettere le modifiche subite dai dati di origine.

### 3.3.3 ZooKeeper

Un'applicazione BigIndex memorizza i dati di configurazione come quelli relativi ai modelli di entità in una repository di dati in rete, servendosi di ZooKeeper.

Apache ZooKeeper è un progetto Apache open source che fornisce un servizio altamente affidabile e sincronizzato per la configurazione di dati quali, a titolo di esempio, quelli richiesti su sistemi distribuiti su larga scala. I dati di configurazione specifici dell'applicazione sono memorizzati in diversi *namespace*, consentendo ad un'unica installazione di ZooKeeper di supportare simultaneamente requisiti di configurazione di più applicazioni.

I requisiti di *reliability* e *availability* vengono soddisfatti da ZooKeeper coordinando i suoi dati su più sistemi, in relazione tra di loro, sui quali è eseguito un server ZooKeeper e che supportano lo stesso namespace per i dati di configurazione.

Un insieme di nodi ZooKeeper che interagiscono tra di loro sono noti con il nome di *ensemble* (o cluster). Configurare un numero di nodi dispari fa sì che un subset di server utilizzi un meccanismo di *majority voting* per selezionare un membro dell'ensemble come il *master authoritative node*. Gli aggiornamenti per qualsiasi membro del gruppo sono sincronizzati con il master, che quindi aggiorna opportunamente tutti gli altri nodi.

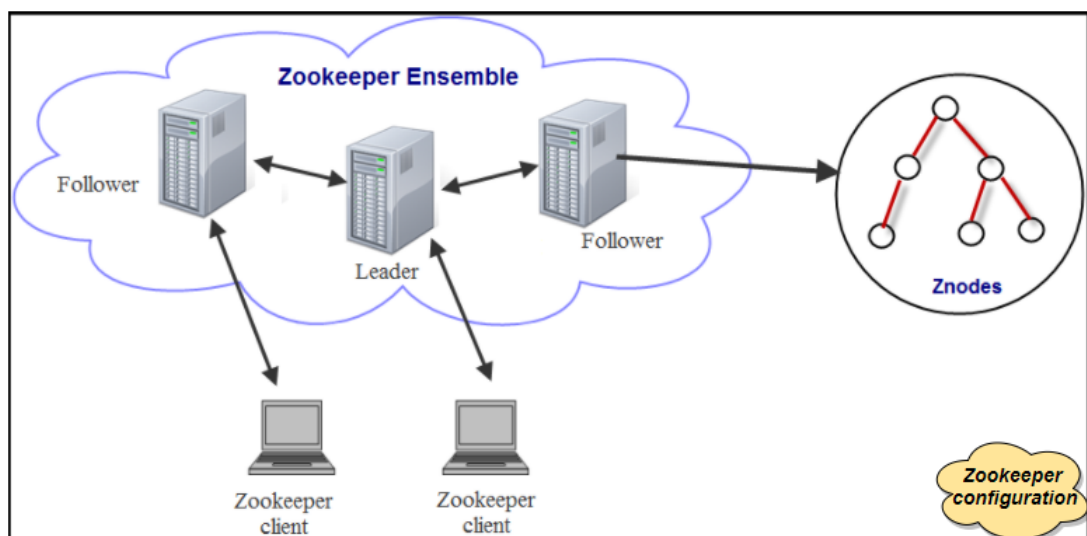


Figura 3.5: Architettura fisica di ZooKeeper

### 3.4 Piattaforma assistita di ricerca

L'utente finale fruisce dell'attività di ricerca grazie alla predisposizione di un *search engine* nel portale web della piattaforma. Il motore di ricerca reso disponibile dall'interfaccia si occupa di prendere la query immessa dall'utente e l'insieme delle opzioni, eventualmente selezionate, di raffinamento della ricerca, e costruire la chiamata da inoltrare al Search Layer. Quest'ultimo ritornerà un set di risultati (eventualmente vuoto) che saranno poi visualizzati nell'interfaccia servendosi di un meccanismo di paginazione dei risultati stessi. La scelta di dotare la piattaforma di ricerca di un form di arricchimento (opzionalmente compilabile, in maniera parziale o in toto), rappresentante l'insieme delle feature disponibili, è giustificata dal voler rendere il più completa e al contempo user-friendly possibile la modalità d'interazione tra utente e search-engine.

Nell'ottica di offrire un servizio che vada incontro a quelle che potrebbero essere le esigenze dell'utente finale, l'attività di ricerca è guidata da un assistente digitale di supporto integrato nella piattaforma, che interagisce con l'utente elaborando le sue richieste espresse in linguaggio naturale.

#### 3.4.1 IBM Watson Conversation

Il servizio Conversation<sup>3</sup> consente di creare una soluzione in grado di comprendere l'input in linguaggio naturale e utilizzare il Machine Learning per rispondere ai clienti, in modo da simulare una conversazione tra gli esseri umani.

L'architettura generale[20] di una soluzione sviluppata servendosi dello strumento prevede che:

1. l'utente interagisca con l'applicazione attraverso l'interfaccia implementata, ad esempio una finestra di chat, un'applicazione mobile o un bot con interfaccia vocale
2. l'applicazione invii l'input al Conversation, connettendosi ad uno spazio di lavoro o *workspace*, che è un contenitore per il flusso di dialogo e i dati di addestramento; il servizio interpreta poi l'input fornitogli (eventualmente sfruttando, per l'analisi, ulteriori servizi Watson come *Tone Analyzer* o *Speech to Text*) e dirige opportunamente il flusso della conversazione, raccogliendo le informazioni di cui necessita

---

<sup>3</sup><https://www.ibm.com/watson/services/conversation/>

3. l'applicazione interagisca, opzionalmente, con un sistema di back-end (come i database dei clienti o i sistemi di elaborazione dei pagamenti) in base all'intento dell'utente e ulteriori informazioni

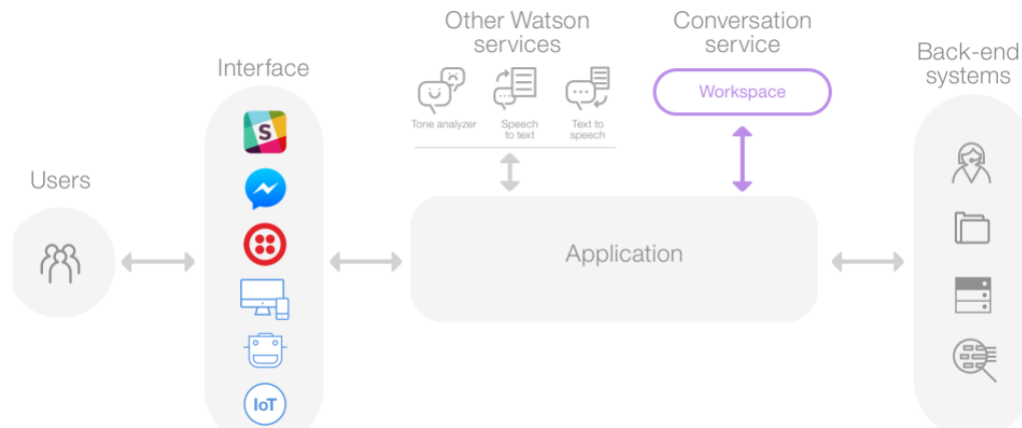


Figura 3.6: Architettura generale di una soluzione che sfrutta il Conversation

Per quanto riguarda i dati di addestramento, questi sono costituiti dagli *Intenti* e dalle *Entità*.

Gli **Intenti** sono gli obiettivi che si prevede abbiano gli utenti all'atto dell'interazione con il servizio. Si suole definire un intento per ogni obiettivo che può essere identificato nell'input utente. Scegliere l'intento corretto per l'input dell'utente è il primo passo per fornire una risposta utile.

L'**Entità**, invece, rappresenta un termine o un oggetto nell'input utente che fornisce chiarimenti o specifici contesti per un particolare intento. Un'entità definisce una classe di oggetti, con valori specifici che rappresentano possibili oggetti in quella classe.

In altri termini, se gli intenti rappresentano i verbi (un qualcosa che un utente vuole fare), le entità rappresentano i nomi (come l'oggetto o il contesto per un'azione). Ciò vale a dire che le entità consentono a un singolo intento di rappresentare più azioni specifiche.

Man mano che nuovi dati di addestramento vengono aggiunti, un classificatore di linguaggio naturale viene automaticamente inserito nello spazio di lavoro ed addestrato a comprendere le tipologie di richiesta per le quali si è indicato che il Conversation debba ascoltare e fornire una risposta.

Lo strumento di **Dialogo**, infine, viene utilizzato per creare un flusso di dialogo che incorpori gli intenti e le entità. Graficamente, lo strumento di Dialogo è rappresentato come una struttura ad albero, ove è possibile aggiungere un ramo per l'elaborazione di ciascuno degli intenti che si desidera che il servizio gestisca. In aggiunta, è possibile inserire dei nodi di diramazione che

gestiscono le numerose possibili permutazioni di una richiesta in base ad altri fattori, come le entità trovate nell'input utente o informazioni passate al servizio dall'esterno.

### 3.4.2 Node-RED

La modalità d'interazione tra l'assistente digitale e l'insieme dei servizi integrati nella conversazione tra l'utente finale e il chatbot viene gestita servendosi di un linguaggio di programmazione grafico chiamato Node-Red<sup>4</sup>, uno tra i più noti tool di *flow-based programming* per l'Internet of Things (IoT).

L'obiettivo di Node-Red è quello di dare a tutti, esperti di programmazione e non, la possibilità di collegare tra di loro diversi dispositivi, oltre ad API e servizi online, in modo da poter realizzare in maniera semplice ed intuitiva dei sistemi altamente integrati e complessi.

Node-RED[21] è interamente scritto in JavaScript e gira su Node.js, la nota piattaforma basata sul JavaScript Engine V8 di Google per la realizzazione di applicazioni server-side. Tra i vantaggi legati all'utilizzo di Node-Red vanno sicuramente citati i seguenti:

- possibilità di sviluppare soluzioni scalabili ed efficienti per l'analisi real-time di flussi di dati, grazie all'adozione del modello event-driven ereditato da JS e all'esecuzione asincrona dell'I/O
- possibilità di servirsi di *package* già pronti e disponibili, semplicemente da installare e aggiungere al flusso di lavoro

---

<sup>4</sup><https://nodered.org/>

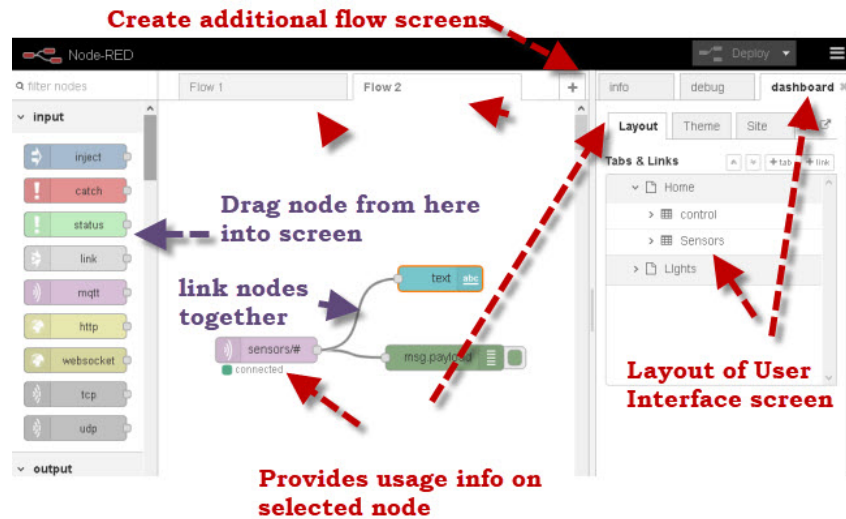


Figura 3.7: Web-Interface di Node-Red per la creazione dei flussi

Un'applicazione Node-Red è definita come **flow** e il codice JavaScript è presente sia nel run-time che garantisce l'esecuzione di uno o più flussi che nei blocchi che li compongono (definiti *nod*i). Ciascuno dei nodi ha, dunque, una parte che potrebbe definirsi *code-behind*, che contiene l'effettiva implementazione del processo e l'elaborazione eseguita dal nodo in oggetto. L'altra parte del nodo, per quanto riguarda gli aspetti visuale e di configurazione, è descritta tramite codice HTML, e anche qui viene fatto uso di JavaScript come linguaggio client-side per definire la *user-experience*. Le connessioni tra i diversi nodi definiti in un flow sono realizzate attraverso le *porte* di input e di output di cui ciascun nodo è dotato. Ciascun flusso è realizzabile in maniera immediata ed intuitiva servendosi del tool visuale (*browser-based UI*), oppure è possibile scrivere con un comune editor di testo il codice relativo al flusso, dal momento che quest'ultimo risulta descritto essenzialmente da un file in formato JSON.

# Capitolo 4

## Implementazione del sistema

*Nel presente capitolo ci si addentrerà nel pieno dello sviluppo e dell'implementazione della soluzione. Per ciascuna delle macro-parti ne verrà descritta e dettagliata l'implementazione, assieme all'utilizzo che ne è stato fatto, in queste, degli strumenti e delle tecnologie presentate nel capitolo precedente.*

*Saranno presentati anzitutto i dettagli relativi alla costruzione e all'indicizzazione dei documenti, e il meccanismo messo in atto per l'aggiornamento degli indici corrispondenti. Dopodiché verrà fornita una descrizione del layer di ricerca e, infine, della piattaforma web sviluppata per l'interfacciamento dell'utente finale con il sistema. Di pari passo sarà presentato il dettaglio relativo all'integrazione dell'assistente digitale nel portale di ricerca.*

### 4.1 Generazione e mantenimento degli indici

Gli applicativi di back-end che si occupano della gestione degli indici associati ai documenti sono realizzati servendosi di SpringBoot oltre che, naturalmente, delle API messe a disposizione da BigIndex.

Spring è un framework applicativo costruito su Java che permette la realizzazione di sistemi disaccoppiati mediante la *dependency injection*. Lo strumento di cui si è fatto uso, Spring Boot, rende più semplice la creazione di applicazioni Spring richiedendo quasi nessuna configurazione: cose come le versioni di libreria vengono automaticamente risolte e l'implementazione è semplicissima. Fondamentalmente, SpringBoot raddoppia la semplificazione dello sviluppo Java di Spring.

Per l'interfacciamento con il database relazionale si è fatto uso di **Spring Data JPA**, una libreria 'helper' di Spring che migliora significativamente l'implementazione dei livelli di accesso ai dati, riducendo al minimo lo sforzo implementativo effettivamente necessario. Allo sviluppatore è richiesto di

scrivere l'interfaccia che fornisce l'insieme delle operazioni CRUD per le entità, in corrispondenza uno ad uno con l'entità nel modello relazionale dei dati, delle quali Spring fornirà automaticamente l'implementazione. In aggiunta, all'esigenza risulta possibile definire delle operazioni custom, delle quali bisognerà però fornire la specifica implementazione.

Il primo dei due applicativi sviluppati si occupa dell'attività di Initial Load dei dati, ovverosia della *creazione* di una collezione e successivo *load* dei documenti indicizzati. Il secondo applicativo, invece, consente di simulare il meccanismo, azionato al verificarsi di una qualche modifica subita dai dati originari, di ricostruzione dell'indice associato ai dati in oggetto.

### 4.1.1 Configurazione Watson Explorer Engine

La struttura dati che viene utilizzata come contenitore dei documenti indicizzati prende il nome di *search collection*. Per creare una search collection, dopo aver eseguito il login nel tool di amministrazione di Watson Explorer Engine, si accede al tab 'Search collections' presente sul lato sinistro dell'interfaccia e, nel dettaglio, al pannello di creazione di una nuova search collection.

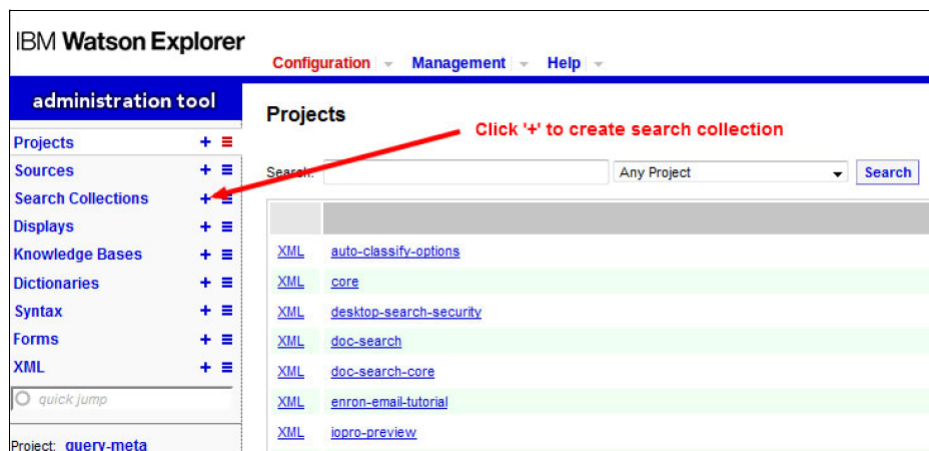


Figura 4.1: Creazione di una search collection su WEX

Qui è possibile inserire il nome della collezione che si intende creare e selezionare eventualmente, tramite l'opzione *Copy defaults from*, una collezione già esistente da cui ereditarne i settaggi di configurazione. Quest'opzione può essere utile nel caso in cui si necessiti di un insieme di search collection che richiedono una configurazione simile (o identica).

Una volta creata la collezione, lo step successivo è quello di configurarla di modo che sia associata ad una specifica repository di dati. È possibile far uso di Watson Explorer Engine per eseguire dapprima il *crawl* dei dati e poi

l'*indexing* degli stessi. In alternativa, ed è l'approccio che è stato seguito nel progetto realizzato, è possibile servirsi delle API BigIndex per bypassare le attività di crawl e indicizzazione messe a disposizione da WEX e gestire *data ingestion* (recupero dei dati) e *data indexing* (indicizzazione dei dati) per via programmatica, accelerandone visibilmente i tempi richiesti.

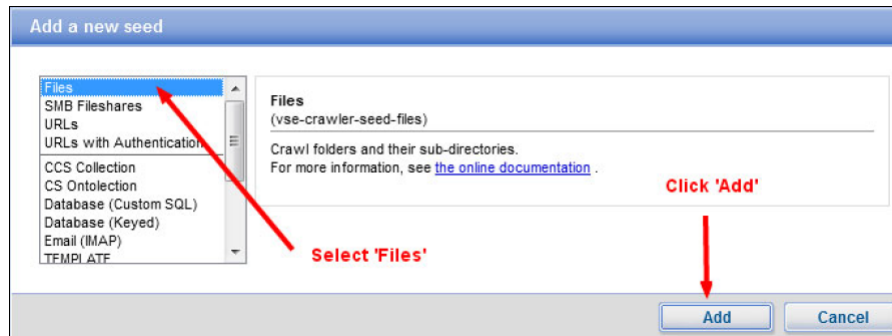


Figura 4.2: Configurazione di un seed - WEX Search collection

Se si sceglie di fare pieno affidamento su Watson Explorer Engine, il passaggio successivo alla creazione della collezione coincide con l'aggiunta di un *seed*. Un seed rappresenta un entry-point che Watson Explorer Engine può utilizzare al fine di estrarre ricorsivamente i dati da una risorsa remota. Una search collection può avere diversi seed, che possono anche puntare a repository diverse. Nella maggior parte dei casi si suole creare una nuova search collection per ciascuna nuova repository della quale è da eseguire il crawling. Watson Explorer Engine consente, in alternativa, di creare delle search collection che non facciano uso di un seed. Nel dettaglio, invece di eseguire un *pull* del contenuto tramite il crawling della risorsa che si vuole indicizzare, è possibile alternativamente fare in modo che il contenuto da indicizzare vada direttamente ad alimentare il crawler mediante un'attività di *push* eseguibile tramite API. Avendo fatto uso di BigIndex API, infatti, l'interfacciamento con la sorgente dei dati viene gestito direttamente dal layer che si occupa dell'indicizzazione dei documenti.

Nel caso in cui si fosse voluto adottare l'altro approccio sarebbe stato sufficiente, invece, aggiungere alla collezione il seed più appropriato, come ad esempio *Database (Custom SQL)* qualora la sorgente dei dati fosse un database, e in tal caso immettere la query SQL atta a selezionare l'insieme dei record da indicizzare.



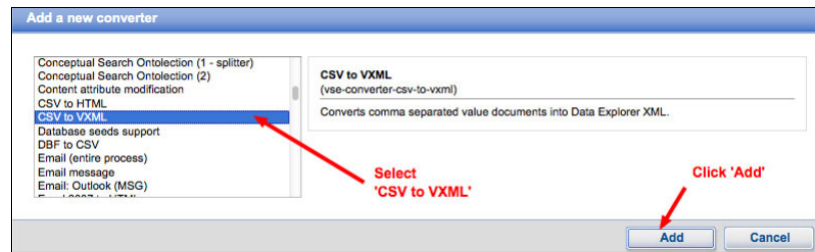


Figura 4.3: Configurazione di un converter - WEX Search collection

In aggiunta sarebbe stata da configurare una *conversion pipeline* al fine di garantire la conversione dei dati (in figura un csv), prima della loro effettiva indicizzazione, nel formato interno VXML utilizzato da WEX.

Gli step immediatamente successivi, indipendentemente dall'approccio di crawling/indexing seguito, sono relativi alla configurazione della search collection per l'utilizzo del *fast-index* su determinati campi e all'abilitazione della *term expansion support*. Il primo dei due step consente di indicare alla collezione di memorizzare gli specifici campi in memoria per il *fast retrieval* e la manipolazione. Il secondo, invece, viene compiuto selezionando l'opzione *Generate dictionaries* e permette di utilizzare l'espansione dei termini in una query, e quindi la ricerca parziale (o per wildcard, più comunemente detta).

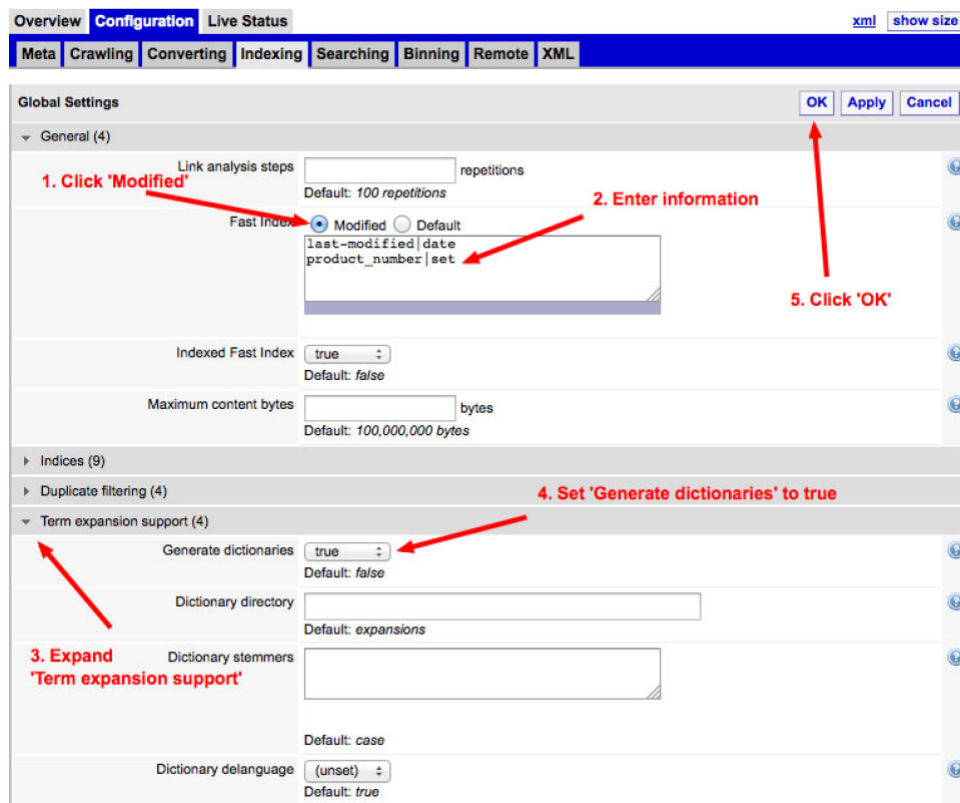


Figura 4.4: Configurazione fast-index e term-expansion

In aggiunta, è possibile configurare la *source* associata alla search collection. Una source individua come una specifica risorsa online è esposta alle operazioni di ricerca, come una query di ricerca viene eseguita sulla risorsa, e il modo in cui l'insieme dei risultati ritornati dalla ricerca devono essere processati. Le source sono tipicamente associate con le search collection di Watson Explorer Engine, ma possono identificare anche altri motori di ricerca o risorse online verso le quali le query di ricerca devono essere indirizzate e dalle quali, quindi, i risultati recuperati. Ciascuna source è composta da un *Form* che consente di definire delle variabili che sono usate all'atto dell'interrogazione della sorgente dei dati ad esso associata.

Nel dettaglio, viene configurata la source per far sì che sia possibile applicare un certo criterio di ordinamento, in fase di ricerca, al set di documenti ottenuto come risultato di una query.

#### Source crm-configuration-driven-indexing-collection\_1\_1

last modified by data-explorer-admin February 6, 2018

Overview Form Parser Declarations Login / Logout Testing XML

VSE Source Form edit remove

Sorting

Sort name-asc|ascending:\$NOME  
name-desc|descending:\$NOME  
surname-asc|ascending:\$COGNOME  
surname-desc|descending:\$COGNOME  
cf\_piva-asc|ascending:\$CF\_PIVA  
cf\_piva-desc|descending:\$CF\_PIVA  
data\_nascita-asc|ascending:\$DATA\_NASCITA  
data\_nascita-desc|descending:\$DATA\_NASCITA

Collection location (6)

Collection name crm-configuration-driven-indexing-collection\_1\_1

Figura 4.5: Configurazione dell'ordinamento dei documenti

Una volta terminata la configurazione della search collection è possibile avviare le attività di crawling e indicizzazione, delle quali potranno essere osservati stato e dettaglio nel tab *Live status*. Se si adotta l'approccio programmatico mediante API BigIndex, invece, dopo aver completato l'operazione di configurazione della collezione è possibile procedere al lancio dell'applicativo che si occupa di reperire ed indicizzare i dati.

### 4.1.2 Initial Load dei dati mediante BigIndex API

**Modello di configurazione dell'applicativo** Un applicativo che fa uso delle BigIndex API adotta uno tra i due approcci disponibili per specificare i dati di configurazione di Watson Explorer Engine.

Il primo approccio prevede che si faccia uso della classe *ExampleBigIndexConfigurationGenerator*, messa a disposizione dal package *bigindex.exampleutils* per

definire una variabile di tipo *Configuration* che agisce da container per le classi *EntityModel*, *DataStores*, e *DataSources*.

---

```
import bigindex.exampleutils.ExampleBigIndexConfigurationGenerator;
Configuration configuration = generateConfiguration();
```

---

Mediante il metodo *generateConfiguration* vengono dunque specificati data source (Watson Explorer Engine instance), data store (collection), e le entity type caratterizzanti il modello. Questo metodo richiama a sua volta il metodo *build* della classe *Builder* di *ExampleBigIndexConfigurationGenerator* al fine di generare la configurazione. L'ultima chiamata è al metodo *getGeneratedConfiguration*, che ritorna la nuova configurazione.

---

```
public static Configuration generateConfiguration() {
    return new
        ExampleBigIndexConfigurationGenerator.Builder(DATA_EXPLORER_ENDPOINT_URL,
            DATA_EXPLORER_ENDPOINT_USERNAME,
            DATA_EXPLORER_ENDPOINT_PASSWORD,
            COLLECTION_NAME, ENTITY_TYPE).build()
        .getGeneratedConfiguration();
}
```

---

La configurazione risultante è passata infine al metodo *build* della classe *ExampleEmbeddedZookeeperServer* in modo da creare un'istanza embedded del server ZooKeeper.

L'approccio seguito nella soluzione, invece, non fa uso della classe *ExampleBigIndexConfigurationGenerator*, ma prevede l'utilizzo di un file di modello di configurazione esterna che specifica data source, data store e le entity type. È la stringa che identifica tale file ad essere passata come argomento all'atto della costruzione dell'istanza del server ZooKeeper.

#### configurationDriven.xml

```
<configuration>
  <data-stores>
    <cluster-collection-store
      collection-name="crm-configuration-driven-indexing-collection"
      name="crm-configuration-driven-indexing-store"
      base-collection="default-big-index-push" total-shards="1">
      <data-source-reference id="instance-1" />
    </cluster-collection-store>
  </data-stores>
  <entity-model>
    <entity-definition name="viewCliente"
      store-name="crm-configuration-driven-indexing-store">
```

```

    <field name="CONT_ID" type="long" searchable="false" retrievable="true"
        sortable="false" />
    <field name="CF_PIVA" type="text" retrievable="true"
        filterable="true" sortable="true" />

<!-- TIPO_ANAGRAFICA, SESSO, COGNOME, NOME, DATA_NASCITA,
    COMPAGNIA, AGENZIA,
    INDIRIZZO, LOCALITA, COMUNE, CAP, PROVINCIA, NAZIONE, POLIZZA,
    DATA_VIGORE, TARGA1, TARGA2 -->

    <field name="LAST_UPDATE_DT" type="text" searchable="false" sortable="true"
        />
</entity-definition>
</entity-model>
<data-sources>
    <data-explorer-engine-instance username="data-explorer-admin"
        url="http://localhost:9080/vivisimo/cgi-bin/velocity" password="TH1nk1710"
        id="instance-1" />
</data-sources>
</configuration>

```

Nella definizione dell'entità *viewCliente* è stato associato a ciascuno dei campi che la caratterizza l'insieme delle proprietà che specificano i tipi di operazioni di ricerca che il particolare campo supporta. Le proprietà del campo indicano come può essere utilizzato e il tipo di informazioni che può rendere disponibili nei risultati di una ricerca.

Proprietà	Descrizione	Valore predefinito
<b>searchable</b>	definisce se il campo può essere cercato dall'utente finale	true
<b>retrievable</b>	definisce se Watson Explorer Engine memorizza una copia del contenuto indicizzato di modo poi da visualizzarne o meno il contenuto in fase di restituzione del risultato di una ricerca	false
<b>sortable</b>	definisce se il campo può essere utilizzato per ordinare i risultati di una ricerca	false
<b>filterable</b>	definisce se il campo può essere utilizzato in interrogazioni di uguaglianza e range di valori	false
<b>weight</b>	definisce la pertinenza del campo quando i suoi contenuti sono abbinati ad una query di ricerca (maggiore è il peso di un campo, maggiore è il suo impatto sulla rilevanza)	peso di 1,0

Tabella 4.1: Proprietà assegnabili ad un campo nella costruzione del record BigIndex

**Configurazione Embedded Zookeeper Server** L'istanza embedded del server Zookeeper è parte integrante dell'applicativo piuttosto che un'istanza a sé stante del server. Questa persiste per la durata dell'applicazione, terminando

ed eliminando i dati che gestisce nel momento in cui si conclude l'esecuzione. Nel dettaglio, per semplicità d'utilizzo, il server embedded di cui ci si è serviti è un server *stand-alone* e *non-clustered*, dunque non istanze multiple, distribuite e persistenti del server, come si suole avere in un ambiente di produzione per la gestione in modo affidabile della configurazione.

Il primo step coincide con l'inizializzazione di un oggetto *configurationProvider* di tipo *ZookeeperBigIndexConfigurationProvider*, che carica una configurazione BigIndex da un server Zookeeper, e dopodiché fa partire l'esecuzione del server stesso. L'oggetto *configurationProvider* incapsula gli URL degli endpoint relativi ai server Zookeeper e il *namespace* di base per i dati usati. La chiamata al metodo *getZookeeperConfiguration* istanzia la configurazione di Zookeeper, che è passata al costruttore della classe *ZookeeperBigIndexConfigurationProvider*.

---

```
import bigindex.exampleutils.ExampleEmbeddedZookeeperServer;

public ZookeeperBigIndexConfigurationProvider configurationProvider = null;
public ExampleEmbeddedZookeeperServer embeddedZkServer = null;

try{
    embeddedZkServer = new ExampleEmbeddedZookeeperServer.Builder(
        "configurationDriven.xml").build();
    embeddedZkServer.startEmbeddedServer();
    ZookeeperConfiguration zkConfig = embeddedZkServer.getZookeeperConfiguration();
    configurationProvider = new ZookeeperBigIndexConfigurationProvider(zkConfig);
}catch (Exception e) {
    e.printStackTrace();
    if (configurationProvider != null) {
        configurationProvider.close();
    }
    if (embeddedZkServer != null) {
        embeddedZkServer.close();
    }
    System.exit(1);
}
```

---

**Initial Load** Come prima operazione, una volta definita la configurazione ed avviata l'istanza del server Zookeeper, è previsto l'Initial Load dei documenti indicizzati. L'applicativo che si occupa di tale operazione gestisce anzitutto il recupero paginato dei record dei clienti dal database, e dopodiché la costruzione delle viste-documento per ciascuno dei clienti individuati. L'insieme delle viste, per un dato cliente, viene costruito andando a recuperare i record delle entità correlate al cliente, qualora presenti, e mettendo assieme tali in-

formazioni e i dati relativi all'anagrafica del cliente stesso.

Nel dettaglio, la particolare vista di un cliente è individuata dalla terna dei valori di (CONT\_ID, COMPAGNIA, AGENZIA), e viene mantenuta una vista 'di gruppo' che consentirà poi la ricerca della sola anagrafica relativa ad un cliente. Al documento di anagrafica viene assegnato un valore fittizio di 'D'-direzione- per i campi di Compagnia e Agenzia: in fase di ricerca, qualora non venissero specificati i due valori di Compagnia e Agenzia, si restringerà il campo d'azione relativamente alle sole anagrafiche dei clienti.

Per tracciare lo stato delle operazioni di indicizzazione è possibile adottare due approcci, funzionalmente equivalenti. Il primo, basato sulla classe *DataIndexer*, utilizza il *polling* basato su di un oggetto di tipo *RequestStatus* per determinare lo stato dell'operazione. Questo approccio può essere molto costoso sia in termini di memoria che di cicli di CPU. Il secondo approccio, che è quello che si è adottato nell'implementazione della soluzione, consente invece di utilizzare le cosiddette *callback routine*: tale approccio rende più semplice ma soprattutto più efficiente l'implementazione, facendo in modo che l'attenzione dello sviluppatore sia rivolta più sul come reagire quando lo stato di un'operazione cambia, e meno su come tenere traccia dello stato. Chiaramente, più grande è il numero di record da indicizzare, più alto è il guadagno potenziale in termini di efficienza. Questo perché, invece di adottare un approccio *poll and block* in attesa che l'operazione legata all'indicizzazione del singolo record termini, l'applicazione è notificata in modo asincrono, per tramite della callback definita, ogni qualvolta sia da segnalare un cambiamento di stato nella singola operazione di indicizzazione.

#### CRM\_InitialLoad-IndexUpdate/Controller.java

```
1 @RequestMapping(value = "/generateViewsClienti/{pageSize}/{numPages}", method =  
    RequestMethod.GET)  
2  
3 public void generate_viewsClienti(@PathVariable int pageSize, @PathVariable int  
    numPages) {  
4     Long contId;  
5     String compagnia;  
6     String agenzia;  
7  
8     RecordBuilderFactory recordBuilderFactory = new  
        RecordBuilderFactory(sti.configurationProvider);  
9     IndexerOptions options = new IndexerOptions.Builder().build();  
10    callbackDataIndexer = new CallbackDataIndexer(sti.configurationProvider, options,  
        uniqueIndexerId);  
11    IndexedCounterCallback callback = new IndexedCounterCallback();  
12    RecordBuilder recordBuilderForEntityType =
```

```

        recordBuilderFactory.newRecordBuilder(ENTITY_TYPE_CLIENTE);
13    try {
14        // Indicizzazione di pageSize clienti alla volta, per un numero di volte pari a
15        // numPages
16        for (int times = 0; times < numPages; times++) {
17            Page<Cliente> page = ClientiRepo.findAllPageable(new
                PageRequest(recordIndex, pageSize));
18            if (page.hasContent()) {
19                for (Cliente c : page) {
20                    // Inizio a costruire la view del cliente impostando gli attributi
21                    // caratterizzanti l'entità Cliente
22                    NormalizedCliente clienteView = new NormalizedCliente(c.getContid(),
                        c.getCfpiva(), c.getTipoanagrafica(), c.getSesso(), c.getCognome(),
                        c.getNome(), c.getData_nascita());
23                    contId = c.getContid();
24                    appLogger.info("Cliente: " + contId);

```

Al servizio che viene esposto è stato assegnato il nome *generate\_viewsClienti*. Il servizio è richiamabile passando come parametri (1) alla chiamata GET la dimensione della pagina e il numero delle pagine da recuperare con la singola chiamata al servizio. Il parametro *pageSize* indica il numero dei record dei clienti da ottenere con la chiamata alla *findAllPageable* (17,18), che ritorna un oggetto di tipo *Page*, corrispondente all'insieme di oggetti Cliente di dimensione pari a *pageSize*.

Un oggetto di tipo *RecordBuilderFactory*(8) produce a sua volta degli oggetti *RecordBuilder*(12), i quali consentono di creare gli effettivi record immagazzinati nell'indice per l'applicazione. L'oggetto di tipo *RecordBuilderFactory* viene istanziato da una chiamata al costruttore della classe, che prende come argomento l'oggetto configurazione indicante lo schema da adottare nella creazione dei *record builder*, mediante l'indicazione del nome dell'entità associata alla tipologia di record. Nel caso in oggetto, l'entità, l'unica in gioco e alla quale viene fatto costantemente riferimento nella costruzione del record da indicizzare, assume il nome di *viewCliente*).

In un ciclo (16), per un numero di iterazioni dettato dal numero di pagine richieste dalla chiamata al servizio, viene costruita la vista del cliente, per ciascun cliente nella pagina recuperata e per ciascuna coppia (Compagnia, Agenzia) presso la quale il dato cliente possiede dei record relativi al Portafoglio e all'Indirizzo.

Nella costruzione della vista ci si serve di una classe custom creata al fine di gestire le informazioni legate all'anagrafica di un cliente, ovvero *NormalizedCliente*(22).



## CRM\_InitialLoad-IndexUpdate/Controller.java

```

25      RecordBuilder recordDIR = recordBuilderForEntityType.id(ID_CLIENTE
      + clienteView.getCONT_ID());
26      if (!clienteView.getCONT_ID().toString().trim().equals("") &&
      clienteView.getCONT_ID() != null)
27          recordDIR.addField("CONT_ID", clienteView.getCONT_ID());
28      if (!clienteView.getCF_PIVA().trim().equals("") &&
      clienteView.getCF_PIVA() != null)
29          recordDIR.addField("CF_PIVA", clienteView.getCF_PIVA());
30      if (!clienteView.getTIPO_ANAGRAFICA().trim().equals("")
31          && clienteView.getTIPO_ANAGRAFICA() != null)
32          recordDIR.addField("TIPO_ANAGRAFICA",
      clienteView.getTIPO_ANAGRAFICA());
33      if (!clienteView.getSESSO().trim().equals("") &&
      clienteView.getSESSO() != null)
34          recordDIR.addField("SESSO", clienteView.getSESSO());
35      if (!clienteView.getCOGNOME().trim().equals("") &&
      clienteView.getCOGNOME() != null)
36          recordDIR.addField("COGNOME", clienteView.getCOGNOME());
37      if (!clienteView.getNOME().trim().equals("") && clienteView.getNOME()
      != null)
38          recordDIR.addField("NOME", clienteView.getNOME());
39      if (!clienteView.getData_NASCITA().trim().equals("") &&
      clienteView.getData_NASCITA() != null)
40          recordDIR.addField("DATA_NASCITA",
      clienteView.getData_NASCITA());
41      recordDIR.addField("COMPAGNIA", "D");
42      recordDIR.addField("AGENZIA", "D");
43
44      Record record_builtedDIR = recordDIR.build();
45      callbackDataIndexer.addOrUpdateRecord(record_builtedDIR, callback);
46      System.out.println("Indexing " + clienteView.getCONT_ID());

```

Il primo passo è quello relativo all'indicizzazione di un documento che rappresenti il cliente nelle sue informazioni condivise a livello di gruppo. Al particolare record da indicizzare viene assegnato un ID corrispondente ad una stringa-identificativo del cliente ("ID\_CLIENTE"), concatenata al CONT\_ID del cliente (25). Ciò consente di avere un documento univoco relativo all'anagrafica di un cliente, poiché questo rappresenterà il cliente nella ricerca dei suoi dati anagrafici.

A questo punto segue l'indicizzazione di ciascuno dei campi caratterizzanti *recordDir*, e per ognuno di essi viene effettuata una chiamata alla *addField*, che aggiunge un campo e il suo rispettivo valore al nuovo record in fase di indicizzazione. Il primo parametro passato non è altro che una stringa che



specifica il nome del campo da aggiungere, mentre il secondo corrisponde al valore del campo indicato e dipende dalla natura del campo specificata nella configurazione xml del modello. Ai campi "COMPAGNIA" e "AGENZIA", infine viene assegnato il valore fittizio di "D".

Il metodo *addOrUpdate* (45) sull'oggetto di classe *CallbackDataIndexer* creato precedentemente è infine utilizzato al fine di indicizzare il record (documento) e aggiungerlo alla collezione. Il metodo aggiunge il record alla collezione dal momento che non è ancora presente, altrimenti avrebbe sortito l'effetto di un aggiornamento, come si vedrà nel seguito.

#### CRM\_InitialLoad-IndexUpdate/Controller.java

```

47         appLogger.info("--- Portafogli cliente ---");
48         List<Portafoglio> portafogliCliente =
            PortafoglioRepo.findByCONT_ID(contId);
49         if (portafogliCliente.size() != 0) {
50             for (Portafoglio p : portafogliCliente) {
51                 // l'agenzia(come oggetto), assieme al cliente, andrà a guidare la
                    costruzione della view
52                 compagnia = p.getCompagnia();
53                 agenzia = p.getAgenzia();
54                 Agenzia agenziaObj = new Agenzia(compagnia, agenzia);
55                 appLogger.info("Compagnia: " + agenziaObj.getCOMPAGNIA() +
                    ", Agenzia: " + agenziaObj.getAGENZIA());
56                 //Costruisco la view del portafoglio associato
57                 //alla coppia (CONT_ID, AGENZIA)
58                 PortafoglioView portafoglioView = new
                    PortafoglioView(p.getIdportafoglio(),
59                     p.getContid(), agenziaObj, p.getPolizza(),
60                     p.getDatavigore(), p.getTarga1(),
61                     p.getTarga2());
62                 appLogger.info("\tPortafoglio: " + portafoglioView);
63                 // Inserisco, con chiave agenzia, il portafoglio
64                 String contains = clienteView.insertIntoPortafoglio(agenziaObj,
                    portafoglioView);
65                 appLogger.info(contains);
66             }
67             List<ViewCliente> listViewsClienti = clienteView.toDocuments();
68             appLogger.info("\tDimensione listViewsClienti: " +
                    listViewsClienti.size());
69             for (ViewCliente vcliente : listViewsClienti) {
70                 RecordBuilder recordSeg =
                    recordBuilderForEntityType.id(ID_CLIENTE +
                    vcliente.getCONT_ID()
                    + "_" + vcliente.getCOMPAGNIA() +
                    vcliente.getAGENZIA());

```

```

71          // un cliente, presso una data (C,A), presenta un INDIRIZZO
              potenzialmente diverso
72      Indirizzo add =
              IndirizzoRepo.findByCONT_ID_C_A(vcliente.getCONT_ID(),
73          vcliente.getCOMPAGNIA(), vcliente.getAGENZIA());
74      if (add != null) {
75          if (!add.getINDIRIZZO().trim().equals("") &&
              add.getINDIRIZZO() != null)
76              recordSeg.addField("INDIRIZZO", add.getINDIRIZZO());
77          if (!add.getLOCALITA().trim().equals("") &&
              add.getLOCALITA() != null)
78              recordSeg.addField("LOCALITA", add.getLOCALITA());
79          if (!add.getCOMUNE().trim().equals("") &&
              add.getCOMUNE() != null)
80              recordSeg.addField("COMUNE", add.getCOMUNE());
81          if (!add.getCAP().toString().trim().equals("") &&
              add.getCAP() != null)
82              recordSeg.addField("CAP", add.getCAP());
83          if (!add.getPROVINCIA().trim().equals("") &&
              add.getPROVINCIA() != null)
84              recordSeg.addField("PROVINCIA", add.getPROVINCIA());
85          if (!add.getNAZIONE().trim().equals("") &&
              add.getNAZIONE() != null)
86              recordSeg.addField("NAZIONE", add.getNAZIONE());
87      } else {
88          appLogger.info("--- NO INDIRIZZO FOUND PER
              CONT_ID: " + contId + " - (C,A) : "
89              + vcliente.getCOMPAGNIA() + "," +
              vcliente.getAGENZIA() + " ---");
90      }
91      if (!vcliente.getPOLIZZA().equals(""))
92          recordSeg.addField("POLIZZA", vcliente.getPOLIZZA());
93      if (!vcliente.getData_VIGORE().equals(""))
94          recordSeg.addField("DATA_VIGORE",
              vcliente.getData_VIGORE());
95      if (!vcliente.getTARGA1().equals(""))
96          recordSeg.addField("TARGA1", vcliente.getTARGA1());
97      if (!vcliente.getTARGA2().equals(""))
98          recordSeg.addField("TARGA2", vcliente.getTARGA2());
99
100      Record record_builted = recordSeg.build();
101      callbackDataIndexer.addOrUpdateRecord(record_builted,
          callback);
102      System.out.println("Indexing " + vcliente.getCONT_ID() + "_" +
          vcliente.getCOMPAGNIA()
103          + vcliente.getAGENZIA());

```

```

104         }
105     } else {
106         appLogger.info("--- NO PORTAFOGLI FOUND PER CONT_ID: " +
            contId + " ---");
107     }
108 }

```

Analogamente, il prossimo step porta all'indicizzazione dei documenti del cliente presso il particolare segmento di agenzia. Ciò che contraddistingue lo step corrente da quello appena conclusosi, è semplicemente l'insieme delle informazioni che vengono indicizzate per la vista.

A partire da un cliente, mediante la sua chiave `CONT_ID`, vengono recuperati i portafogli ad esso associato (48), e per ciascun record di Portafoglio viene dapprima creata un'istanza della classe custom *PortafoglioView* (58), e poi eseguito un check di esistenza sul set di portafogli del cliente per far sì che il dato portafoglio sia inserito nella medesima vista del cliente presso il segmento di agenzia corrispondente.

Mentre *recordDir* rappresentava il cliente in termini di anagrafica, *recordSeg* (69) contiene adesso le informazioni, se disponibili, legate alle entità correlate Portafoglio e Indirizzo. Al documento indicizzato viene assegnato l'identificativo corrispondente alla concatenazione della stringa `ID_CLIENTE`, e dei valori dei campi `CONT_ID`, `COMPAGNIA` e `AGENZIA`.

Il record viene infine costruito (100) e predisposto all'indicizzazione nella collezione (101). Qualora il cliente non presenti alcun portafoglio (105) non viene compiuta alcuna operazione di indicizzazione, e dunque per tale cliente l'unica vista costruita ed indicizzata corrisponde a quella contenente i soli dati anagrafici.

#### CRM\_InitialLoad-IndexUpdate/Controller.java

```

110     try {
111         System.out.println("waitForAllIndexingToComplete in corso...");
112         callbackDataIndexer.waitForAllIndexingToComplete();
113     } catch (IndexerInterruptedException e) {
114         e.printStackTrace();
115         System.exit(1);
116     }
117     System.out.println("success! Done Indexing Data...");
118     System.out.println("*** Indexing results" + " ***");
119     System.out.println("Total Number of Enqueued Records :...." +
        callback.getTotalEnqueued());
120     System.out.println("Total Number of Indexed Records :...." +
        callback.getTotalSuccess());

```

```

121         System.out.println(
122             "Total Number of Partially Indexed Records :...." +
                callback.getTotalPartialSuccess());
123         System.out.println("Total Number of Failures :...." +
                callback.getTotalFailure());
124
125         System.out.println("*---*   *---*");
126     }
127     // Altro giro, altra corsa (pageSize fisso, una volta entrati nel ciclo)
128     recordIndex += pageSize;
129 }
130 } finally {
131     callbackDataIndexer.close();
132 }
133 /*
134  * alla prossima chiamata del servizio, richiedo una pagina di pageSize
135  * records a partire da quello successivo all'ultimo elaborato
136  * recordIndex += pageSize;
137  */
138 }

```

---

La chiamata alla *waitForAllIndexingToComplete* (112) sull'oggetto *callbackDataIndexer* blocca indefinitamente finché l'attività di indicizzazione non viene completata. Una seconda versione del metodo consente di specificare un timeout, scaduto il quale l'attesa viene interrotta e non risulta più possibile riuscire a determinare l'esito dell'indicizzazione. La chiamata viene eseguita all'interno di un blocco try/catch, in modo da rispondere nel modo più appropriato alla eventuale *IndexerInterruptedException* (113) sollevatasi durante l'attesa bloccante.

Al termine dell'indicizzazione ci si può servire della callback per ottenere un recap contenente l'indicazione del numero dei record indicizzati con successo, dei record accodati durante l'operazione, e dei record parzialmente indicizzati o non indicizzati (per via del verificarsi di un errore di qualche tipo). La callback può essere definita nel modo che si ritiene più opportuno, ma un utilizzo intelligente di questo strumento è senza dubbio quello di registrare in un apposito file di log gli identificativi corrispondente agli indici per i quali si sono registrati dei problemi, e quindi rieseguirne in un secondo momento l'indicizzazione mirata.

L'attività di indicizzazione è conclusa da una chiamata al metodo *close* (131), che esegue uno 'shut-down' dell'oggetto *callbackDataIndexer* e dell'insieme dei thread associati a questo e coinvolti nella procedura. Tale chiamata viene inserita nella clausola *finally* per garantire il fatto che questa sia eseguita

sia in caso di fallimento che di riuscita dell'operazione, e che quindi tutte le risorse in gioco siano liberate.

### Collection **crm-configuration-driven-indexing-collection\_1\_1**

*last modified by data-explorer-admin 18 minutes ago*

Overview
Configuration
Live Status

**Seeds**  
Seeds must be configured in the [Seeds](#) section of the crawling configuration. No seeds are currently configured for this search collecti

**Working Copy** ⓘ  
create

**Live Status**

<b>Crawling</b> stop Crawling... Elapsed 0:00:56 Complete 1 Pending 0 Unprocessed URLs 0 Unindexed Error URLs 0 Total Crawl Errors 0 Other 0 Size 16 KB	<b>Indexing</b> restart Active... Elapsed 0:00:55 URLs processed 15 Uncommitted URLs 15 Unmerged segments 0 Indices 3 Documents 3,653 Size 2.1 MB
--	---

Figura 4.6: 'Live status' relativo all'indicizzazione

### 4.1.3 Aggiornamento Near-Online (NOL) dell'indice

I servizi *addOrUpdateByJsonMsg* e *deleteByJsonMsg* si occupano di mantenere la consistenza dei dati tra la collezione dei documenti indicizzati e il database. I due gestiscono, rispettivamente, le due operazioni di creazione/aggiornamento e di cancellazione dell'indice associato al documento nella collezione, a seconda del tipo di evento verificatosi nella base di dati.

Il meccanismo messo in atto simula il comportamento di una coda *IBM MQ*<sup>1</sup>, un contenitore di messaggi tramite cui le applicazioni aziendali, connettendosi al gestore code su cui è presente tale coda, possono richiamare i messaggi o inserire i messaggi nella coda stessa. La simulazione della coda è messa a punto mediante un automatismo per la gestione delle notifiche di eventi su db, grazie all'interazione, basata su messaggi, tra i due applicativi coinvolti nel funzionamento generale del processo di reperimento e indicizzazione dei dati.

<sup>1</sup>[https://www.ibm.com/support/knowledgecenter/it/SSFKSJ\\_8.0.0/com.ibm.mq.explorer.doc/e\\_queues.htm](https://www.ibm.com/support/knowledgecenter/it/SSFKSJ_8.0.0/com.ibm.mq.explorer.doc/e_queues.htm)

L'applicativo descritto nella sezione precedente ha il compito di agire puntualmente sul documento il quale indice sia da costruire/aggiornare o eliminare, previa indicazione ottenuta dall'applicativo che verrà presentato nella sezione corrente. Quest'ultimo rende fruibili le operazioni CRUD (Create, Read, Update, Delete) che sono alla base di tutte le applicazioni orientate ai database. La web application progettata consente di scegliere su quale delle tre entità andare ad operare (Cliente, Portafoglio, Indirizzo).

Per ciascuna entità, che sia essa l'entità primaria Cliente o una tra le due entità secondarie Portafoglio e Indirizzo, è predisposto un form compilabile dall'utente (non un utente finale della piattaforma di ricerca, quanto piuttosto un utente abilitato ad interfacciarsi con la base di dati). Il form consente di agire sul singolo record di una tabella del database in termini di creazione, recupero, modifica, e cancellazione.

#### AngularJS - Spring JPA - DB2 (CRM CRUD)



Select one or more entity between Cliente, Portafoglio, and Indirizzo:

Cliente

#### Cliente

##### Update Cliente by CONT\_ID

Cf\_piva

Tipo\_anagrafica

Sesso

Cognome

Nome

Data\_nascita

Update Cliente

##### Delete Cliente by CONT\_ID

Delete Cliente

##### Retrieve Cliente by CONT\_ID

Get Cliente

##### Insert Cliente (PK CONT\_ID)

Cont\_id

Cf\_piva (required)

Tipo\_anagrafica

Sesso

Cognome

Nome

Data\_nascita

Insert Cliente

#### Portafoglio

#### Indirizzo

Figura 4.7: Interfaccia web per il CRUD e la simulazione della coda

Ciò che viene riprodotto è quanto potrebbe essere ottenuto per mezzo della formulazione, in linguaggio SQL, dell'interrogazione corrispondente alla

singola operazione. L'interfaccia web fornita consente di mascherare all'utilizzatore la complessità che si cela dietro alla formulazione di interrogazioni SQL e rende più rapida ed intuitiva la modalità tramite cui si raggiunge il risultato voluto.

Una volta completata l'operazione di inserimento, modifica o cancellazione dell'entità, lo step immediatamente successivo coincide con l'invio del messaggio di notifica dell'avvenuta operazione. La logica applicativa che si cela dietro all'interfaccia web prevede l'esistenza di un *Controller* per ciascuna delle tre entità. Il controller si occupa di mappare la chiamata al servizio nell'operazione da eseguire sul db e, ad operazione completata, di inviare un opportuno messaggio all'applicativo che gestisce l'indicizzazione dei documenti.

#### CRM\_CRUD\_ClientApplication/ControllerCliente.java

```

1  // Update a Cliente
2  @PostMapping("/clienti/{cont_id}")
3  public Response updateCliente(@PathVariable(value = "cont_id") Long cont_id,
4      @Valid @RequestBody Cliente cliente) {
5      Long autoInc_idEvent = clienteRepo.getMaxLasteventid();
6
7      Cliente clienteToUpdate = clienteRepo.findOne(cont_id);
8      if (clienteToUpdate == null) {
9          return new Response("Error", null);
10     }
11
12     if (! cliente .getCf_piva().equals(""))
13         clienteToUpdate.setCf_piva(cliente.getCf_piva());
14     if (! cliente .getCognome().equals(""))
15         clienteToUpdate.setCognome(cliente.getCognome());
16     if (! cliente .getNome().equals(""))
17         clienteToUpdate.setNome(cliente.getNome());
18     if (! cliente .getSesso().equals(""))
19         clienteToUpdate.setSesso(cliente.getSesso());
20     if (! cliente .getData_nascita().toString().equals(""))
21         clienteToUpdate.setData_nascita(Util.convertDateToStringToTimestamp
22             ( cliente .getData_nascita().toString() ));
23     clienteToUpdate.setLast_update_dt(new
24         Timestamp(System.currentTimeMillis()).toString());
25     clienteToUpdate.setLast_event_id(++autoInc_idEvent);
26
27     Cliente updatedCliente = clienteRepo.save(clienteToUpdate);
28     sendMessage("cliente", "update", autoInc_idEvent.toString());
29     return new Response("Done", updatedCliente);
30 }

```

Nel dettaglio, come mostrato in figura da un esempio relativo alla gestione dell'operazione di Update per l'entità Cliente, viene fatto uso dell'informazione relativa al *cont\_id* per recuperare il record d'interesse (7) in modo da rifletterne le modifiche e inviare il messaggio di notifica. Al record in oggetto viene poi associato un *last\_event\_id*, calcolato come incremento di una unità del massimo valore attualmente presente tra i record dell'entità cliente (5). Questo id tiene traccia delle operazioni di inserimento, modifica, o cancellazione di un record, e viene incrementato ogni qualvolta sia registrato uno di questi eventi. In tal modo, il messaggio sarà strutturato affinché contenga, oltre alle indicazioni sull'entità in gioco e sulla tipologia di operazione, l'informazione relativa all'id dell'evento. Quanto detto vale per le operazioni di inserimento e di modifica di un record, ma ovviamente non trova corrispondenza nell'operazione di cancellazione: essendo il record cancellato non più presente sul database, l'insieme dei documenti il cui indice sarà da aggiornare verrà recuperato servendosi dell'informazione relativa al *Cont\_id* del cliente, qualora il record corrisponda alla sua anagrafica, in aggiunta all'informazione sulla *Compagnia-Agenzia*, qualora il record eliminato sia corrispondente ad un'entità correlata al cliente. In questo modo, sfruttando le API BigIndex, sarà possibile eseguire un'operazione di indicizzazione mirata dei documenti, recuperando l'insieme degli indici che, a seguito dell'evento verificatosi sul db, devono essere ricostruiti o cancellati per mantenere la consistenza dei dati.

Evento sul DB	Composizione messaggio	Invio messaggio	Endpoint richiamato	API BigIndex (aggiornamento indice)
INSERT	<ul style="list-style-type: none"> <li>entity</li> <li>id_evento</li> </ul>	sendMessage (entity, "insert", id_evento)	/addOrUpdateByJsonMsg [POST Message]	addOrUpdateRecord (entity, "ID_CLIENTE" +cont_id)
UPDATE		sendMessage (entity, "update", id_evento)		
DELETE (Cliente)	<ul style="list-style-type: none"> <li>entity</li> <li>cont_id</li> </ul>	sendMessage (entity, "delete", cont_id)	/deleteByJsonMsg [POST Message]	deleteRecord (entity, "ID_CLIENTE" +cont_id+C+A)
DELETE (Portafoglio, Indirizzo)	<ul style="list-style-type: none"> <li>entity</li> <li>cont_id</li> <li>+compagnia+agenzia</li> </ul>	sendMessage (entity, "delete", cont_id+C+A)		

Tabella 4.2: Meccanismo di notifica eventi per la consistenza dei dati

Il meccanismo che prevede la notifica immediata, o quasi, in forma di messaggio generato dall'applicativo CRUD a seguito di un evento di Insert, Upda-



te o Delete di un record, prende il nome di aggiornamento *near-online*(NOL), dal momento che la modifica avvenuta sul db si riflette pressoché istantaneamente sulla collezione indicizzata dei documenti. Mediante l'adozione di un approccio NOL è garantito che la consistenza tra le due parti sia costantemente mantenuta, ma è presente il limite di gestire la notifica di un solo evento per volta. In alternativa potrebbe essere utilizzato un approccio *batch*, che consentirebbe sì l'accorpamento di più eventi in una singola notifica ma che ritarderebbe inevitabilmente la sincronizzazione tra il database e la collezione dei documenti, dato che il batch è un'attività non eseguita nell'immediato ma rimandata nel tempo. L'invio in batch di chiamate a un servizio remoto è comunque una strategia nota per migliorare le prestazioni e la scalabilità di un sistema: ogni interazione con un servizio remoto comporta infatti costi fissi di elaborazione. L'approccio scelto e qui descritto è il NOL, che si è ritenuto essere l'approccio ideale per la soluzione realizzata, soprattutto per via del fatto che la frequenza caratterizzante il verificarsi di un evento sul db non è esageratamente elevata. Nulla vieta, però, di accostare al NOL un'attività di aggiornamento batch riuscendo a sfruttare, in tal caso, i vantaggi dell'uno e dell'altro approccio.

## 4.2 RESTful Search Layer

Il macroblocco relativo al *Search Layer* rappresenta la componente della soluzione che si occupa di gestire ciascuna richiesta di interrogazione sulla collezione dei documenti, facendosi carico di prendere, di volta in volta, la query di ricerca dell'utente e l'insieme delle features selezionate dall'utente all'atto della richiesta, ed interrogare la collezione indicizzata mediante l'opportuno uso dell'API SOAP di ricerca *query-search* resa disponibile da Watson Explorer Engine.

Nel dettaglio, il search layer realizzato espone un insieme di chiamate (o servizi) RESTful<sup>2</sup> che hanno a fattore comune lo scopo di prelevare la query di ricerca, stabilire se la ricerca stessa avvenga sui soli dati relativi all'anagrafica o se sul particolare segmento di agenzia, individuare le opzioni eventualmente selezionate, e fornire in risposta l'insieme dei documenti corrispondenti al risultato, paginato, dell'interrogazione costruita sulla base del form riempito dall'utente. La paginazione viene eseguita nel search layer e poi sfruttata nel frontend della piattaforma di ricerca per far sì che l'insieme dei risultati

---

<sup>2</sup>REpresentational State Transfer: uno stile architetturale per sistemi software distribuiti, alla cui base vige il concetto di 'risorsa' accessibile e trasferibile tra client e server

sia visualizzato per pagine di dieci documenti (dieci è il valore che, di default, viene considerato da WEX). In questo modo, la chiamata alla search viene eseguita, all'esigenza, più e più volte, richiedendo di volta in volta la pagina successiva a quella ottenuta mediante l'ultima invocazione. Il meccanismo legato alla paginazione dei risultati consente, quindi, di trarne dei benefici in termini sia di visualizzazione dei risultati stessi per l'utente finale, che potrà muoversi più agevolmente tra i documenti ottenuti, sia in termini di riduzione del carico di elaborazione del server, come anche del client, che si vedrebbe altrimenti arrivare una quantità di risultati interminabile e difficilmente gestibile in tempi ragionevoli.

GET	/search/{area}	servizio di ricerca di informazioni condivise a livello di gruppo per crm search
GET	/search/{area}/{compagnia}	servizio di di ricerca sulla compagnia per crm search
GET	/search/{area}/{compagnia}/{agenzia}	servizio di di ricerca sulla compagnia e agenzia per crm search
GET	/search/filter	restituisce la lista di filtri possibili
GET	/search/fields	restituisce la lista di campi su cui effettuare la ricerca
GET	/search/sortBy	restituisce i tipi di ordinamento possibili

Figura 4.8: Servizi esposti dal Search Layer

I servizi di ricerca implementati prevedono che la chiamata sia effettuata in GET e segua una dei formati indicati in figura. Il path fisso (*search/*) viene seguito dall'indicazione dell'area (*clienti*), opzionalmente da compagnia e/o agenzia (se si intende ricercare il cliente presso il segmento di agenzia). Per completezza, come quadro completo dei servizi fruibili per tramite del search layer, sono presentati in figura anche i servizi che restituiscono l'insieme dei filtri, dei campi, e degli ordinamenti dei quali è possibile servirsi.

<b>q</b> <i>* required</i> string (query)	query di ricerca utente	<b>wildcard</b> string (query)	utilizzato per attivare la funzionalità di ricerca avanzata per prefisso di ognuno (o parte dei) token inserito(i) in query. Per esempio la ricerca 'gian 329' (wildcard su ambo i token) diventa 'gian* 329*'. I valori possibili sono [pre, post, pre-post]
<b>page</b> <i>* required</i> integer (query)	Numero di pagina richiesto	<b>filter</b> string (query)	utilizzato per richiamare un insieme di filtri predefiniti metalinguaggio: ' nome_filtro[:nome_filtro]*'
<b>pageResult</b> integer (query)	Numero di risultati per pagina	<b>fields</b> <i>* required</i> string (query)	utilizzato per definire un insieme di campi su cui effettuare la ricerca metalinguaggio: ' nome_campo[^priorità] [:nome_campo[^priorità]]*'
		<b>sortBy</b> string (query)	utilizzato per richiamare un insieme predefinito di ordinamenti dei risultati di ricerca metalinguaggio: ' nome_ordinamento[:nome_ordinamento]*'

Figura 4.9: Parametri richiesti dai servizi del SL

Il servizio di ricerca delle informazioni condivise sul cliente prevede come unico dato immesso dall'utente la query di ricerca, mentre le informazioni obbligatorie, a livello di costruzione della chiamata al servizio, comprendono anche il numero di pagina richiesto e l'insieme dei campi del documento sui quali debba insistere la ricerca. È l'utente a scegliere, opzionalmente (spuntando semplicemente delle checkbox), se restringere o meno il campo di azione della ricerca ad un sottoinsieme preciso di campi.

```
fieldsResponse v [
example: List [ "CONT_ID", "TIPO_ANAGRAFICA", "NOMINATIVO (NOME + COGNOME)", "CF_PIVA", "SESSO", "DATA_NASCITA", "COMPAGNIA", "AGENZIA", "INDIRIZZO", "LOCALITA", "COMUNE", "CAP", "PROVINCIA", "NAZIONE", "POLIZZA", "DATA_VIGORE", "TARGA1", "TARGA2" ]string]
```

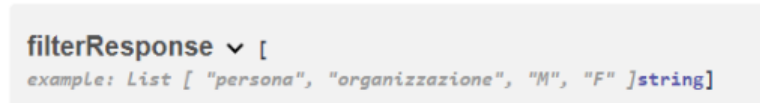
Figura 4.10: Informazioni sul cliente ricercabili dall'utente finale

Qualora tale scelta non venisse effettuata, la ricerca avverrebbe automaticamente sull'insieme completo dei campi ricercabili. Opzionale è anche la selezione delle feature di filtro, ordinamento, e ricerca parziale dell'informazione: la ricerca mirata delle informazioni, assieme alle feature appena citate, contraddistinguono una modalità di *ricerca avanzata*. Mentre la *ricerca base* prevede che sia unicamente fornita la query di ricerca, la modalità avanzata arricchisce tale query e meglio si adatta ad una ricerca puntuale dell'utente, dal momento che la selezione di una o più feature consente a quest'ultimo di raffinare la ricerca e al contempo fruire di funzionalità delle quali, altrimenti, non si riuscirebbe a giovare mediante una semplice ricerca.

I campi del documento che sono stati indicizzati come *fast-indexed* possono

essere utilizzati per filtrare, ordinare, ed eventualmente raggruppare i risultati. La query che viene inviata a Watson Explorer Engine può essere arricchita dunque da condizioni quali filtri ed ordinamenti, che è possibile passare tramite specifici parametri CGI<sup>3</sup>, i quali rivestono un significato particolare per WEX a livello di API.

Una prima feature fruibile è quella del filtro.



```
filterResponse v [
example: List [ "persona", "organizzazione", "M", "F" ]string]
```

Figura 4.11: Filtri applicabili alla ricerca

Risulta possibile restringere il campo d'azione della ricerca su di un insieme di documenti che soddisfino una particolare condizione, ossia quella espressa dal filtro.

Un filtro altro non è che una semplice condizione ' $\langle \text{campo} \rangle = \langle \text{valore} \rangle$ '. Due sono i filtri che è possibile applicare alla ricerca sui clienti. Il primo è relativo al voler eseguire la ricerca su di una persona fisica (filtro 'PERS') o su di un'organizzazione (filtro 'ORG'). Chiaramente, non applicando alcun filtro sulla tipologia di cliente, la ricerca viene estesa sia alle persone che alle organizzazioni. Un secondo filtro consente invece di filtrare per sesso, e rispettivamente restringere la ricerca ai clienti (ovviamente persone fisiche e non organizzazioni) di sesso maschile (filtro 'M') o femminile (filtro 'F').

Per quanto riguarda, invece, l'ordinamento dei documenti restituiti come risultato di una ricerca, il concetto chiave è quello dello score: la cosiddetta *result relevance* si ribadisce essere la misurazione predefinita utilizzata per determinare l'ordine in cui vengono visualizzati i risultati della ricerca. Watson Explorer Engine utilizza una formula per il calcolo della rilevanza che viene utilizzata per calcolare il peso o l'importanza di un risultato di ricerca: nel dettaglio, viene fatto uso delle due variabili predefinite di *score* e *la-score*, indicanti rispettivamente il punteggio di rilevanza del documento rispetto alla query dell'utente e il punteggio calcolato dallo step di *link-analysis* durante la costruzione dell'indice. Nel lavoro di tesi è stato adottato lo score nativo di WEX, per via del fatto che non si è ritenuto necessario apportarne delle modifiche per una qualche ragione. Risulta tuttavia possibile manipolare la formula e adattarla alle proprie esigenze.

L'ordinamento, in generale, è configurato per essere controllato dal campo

---

<sup>3</sup>Common Gateway Interface, una tecnologia standard utilizzata dai Web Server per interfacciarsi con applicazioni esterne

*sortBy*. Di default, come indicato, l'ordinamento previsto è per score decrescente, ma è possibile adottare un criterio di ordinamento che non sia basato sulla rilevanza, quanto piuttosto su criteri dipendenti dai valori assunti da uno o più campi qualora, come nel caso in oggetto, si sia mantenuta una struttura nel documento indicizzato. Un ordinamento potrebbe essere in alternativa previsto per i valori assunti dai metadati che è possibile associare (e quindi indicizzare) ad un particolare documento. Gli ordinamenti sono da specificare all'atto della configurazione della source associata alla collezione dei documenti (fig. 4.5), quindi prima che sia avviata l'effettiva indicizzazione degli stessi. I meccanismi per l'ordinamento dei risultati di ricerca vengono in genere identificati fornendo il parametro '*sortBy=<sort condition>*' nella query di ricerca su WEX, aggiungendo quindi alla keyword *sortBy* l'indicazione del criterio di ordinamento al quale è stato precedentemente assegnato un nome rappresentativo. Risulta possibile ordinare per i valori assunti da un campo (o metadato), sia in modo crescente che decrescente, e il nome che viene assegnato alla condizione di ordinamento è un nome puramente arbitrario.

```
sortByResponse ▾ [  
  example: List [ "name-asc", "name-desc", "surname-asc", "surname-  
    desc", "cf_piva-asc", "cf_piva-desc", "data_nascita-asc",  
    "data_nascita-desc" ]string]
```

Figura 4.12: Servizi esposti dal Search Layer

I criteri di ordinamento resi disponibili nella piattaforma di ricerca del progetto sono relativi al nome, al cognome, al codice fiscale o partita IVA, e alla data di nascita del cliente. Un particolare è rappresentato dalla data di nascita che, non essendo un campo puramente testuale alla pari degli altri tre, è stato indicizzato indicando la natura '*date*' del campo (al passo in fig. 4.4), di modo poi che sia possibile ordinare i valori del campo secondo un ordinamento per data.

Le piattaforme di ricerca come Watson Explorer Engine offrono inoltre tecniche specifiche per il linguaggio in grado di analizzare ed espandere le query in modo che possano corrispondere ad una più ampia gamma di risultati pertinenti. Queste tecniche includono *depluralizing* (rimozione dei suffissi che differenziano tra termini singolari e plurali), *delanguaging* (che normalizza i sistemi di scrittura giapponesi e rimuove i segni diacritici specifici di un linguaggio), e *stemming* (capacità di individuare i risultati di una ricerca che corrispondono alla stessa forma base dei termini nella query).

Oltre ad utilizzare questi tipi di analisi linguistica e dei caratteri, Watson Explorer Engine offre ulteriore flessibilità consentendo di cercare termini specificando un particolare modello da abbinare, piuttosto che un semplice termine. Queste tecniche, generalmente definite di *term expansion*, sono le seguenti:

**wildcards** caratteri speciali (il punto interrogativo (?) e asterisco o stella (\*)) che possono essere utilizzati per rappresentare rispettivamente una sequenza singola o multi-carattere. Il '?' corrisponde a qualsiasi singolo carattere, mentre '\*' corrisponde a qualsiasi sequenza di zero o più caratteri

**regular expressions** forniscono un meccanismo più potente per la corrispondenza dei modelli, consentendo di limitare le corrispondenze del modello a valori di caratteri specifici, intervalli specifici e/o numeri di caratteri, posizioni di caratteri specifici all'interno di un termine, e così via

L'espansione dei termini dev'essere esplicitamente abilitata in ciascuna collezione in cui si desidera poter utilizzare i caratteri jolly e le espressioni regolari (fig. 4.4).

L'insieme dei servizi RESTful che viene esposto dal Search Layer prevede, in corrispondenza di una qualsivoglia tipologia di query immessa, un output formattato come illustrato in figura.

```
searchResponse {
  page          integer
                Numero di pagina attualmente
                restituita

  totalResult   integer
                Numero totale di risultati recuperati
                dal motore di ricerca

  resultInPage  integer
                Numero di risultati presenti nella
                pagina

  previous      string
                url di ricerca per la pagina
                precedente

  next          string
                url di ricerca per la pagina
                successiva

  results       > [...]
```

Figura 4.13: Formato del JSON restituito da un servizio in corrispondenza di una ricerca

Nella risposta è presente l'indicazione sia dei risultati totali prodotti dalla ricerca che dei risultati correnti in pagina, valore, quest'ultimo, limitato superiormente ai 10. L'output contiene l'url alla pagina precedente e a quella successiva, qualora l'insieme dei risultati sia navigabile tra più pagine. L'array *results* contiene la lista dei documenti restituiti nella pagina.

### 4.3 Piattaforma di ricerca assistita

L'ultimo dei tre macroblocchi presentati corrisponde alla parte del lavoro più vicina all'utente finale. Mentre le tre parti precedenti di *indexing* e *searching*, come visto, gestiscono rispettivamente la predisposizione dei dati alla ricerca e l'esposizione dei servizi di ricerca di modo che questi siano fruibili dalla piattaforma, il macroblocco che viene presentato in questa sezione modella l'interfaccia utente del motore di ricerca Google-like ed integra, nell'interfaccia stessa, il tool dell'assistente digitale a supporto dell'attività di ricerca dei clienti. Il frontend è costruito servendosi anche qui del framework AngularJS, mentre per il modello del chatbot si è utilizzato il tool IBM Bluemix Conversation e ci si è serviti di Node-Red per l'integrazione del servizio in un'interfaccia in stile chat.

Figura 4.14: Form compilabile nel portale di ricerca

#### 4.3.1 Portale di ricerca dei clienti

Il form di ricerca che viene presentato all'utente consente a quest'ultimo di scegliere se, e come, servirsi dell'insieme delle funzionalità associabili alla ricerca descritte nella sezione dell'implementazione del Search Layer.

La prima opzione che l'utente può selezionare è relativa alla modalità con cui egli intende ricercare i termini inseriti nella query: in alternativa al classico

*AND* dei termini è possibile fare in modo che la ricerca avvenga in *OR*. Di default, se non viene spuntato nulla relativamente a tale opzione, un documento apparirà tra i risultati solo ed esclusivamente se al suo interno figurano ciascuno dei termini componenti la query.

In aggiunta, l'utente può voler dare un peso maggiore nella ricerca a particolari termini, e tale esigenza può essere soddisfatta inserendo nell'apposita casella di testo il termine o i termini che si vuole prevalgano nel reperimento dei risultati. Ad esempio, se l'utente inserisce la query 'Mario Rossi', abilitando la ricerca in *OR* e specificando la priorità su 'Rossi', viene ottenuta la seguente query di ricerca: 'Mario OR Rossi^10', dove 10 è un peso che indica a Watson Explorer Engine che quel termine debba avere un peso maggiore sui restanti. In questo modo, tra i risultati della ricerca, ai documenti contenenti 'Rossi' verrà assegnato dal motore di ricerca uno score maggiore e, di conseguenza, nell'ordinamento per rilevanza, occuperanno una posizione 'migliore' rispetto agli altri.

La sezione successiva permette, come detto in più di un'occasione, di far agire la ricerca solo su determinati campi. Può essere utile, ad esempio, nella ricerca di un nominativo, restringere la ricerca al solo campo nominativo, e quindi evitare di ottenere dei documenti 'spuri' nel risultato. Può accadere che nel risultato siano presenti dei documenti che hanno sì avuto un match con la query dell'utente, ma lo hanno avuto su di un campo diverso dal nominativo, come ad esempio indirizzo.

I due box 'Compagnia' e 'Agenzia' permettono all'utente di ricercare presso un determinato segmento di agenzia e quindi ottenere, in corrispondenza della query di ricerca, l'insieme dei dettagli del cliente legati al segmento, ovvero indirizzo, polizze, e così via. Qualora i due box non vengano riempiti, la ricerca agirà sulle informazioni condivise tra le agenzie, ovvero sulle anagrafiche.

Di seguito l'utente ha la possibilità di scegliere se applicare alla ricerca uno dei due filtri, o se ordinare i risultati ottenuti per un particolare criterio (ascendente o discendente). Risulta possibile uno o entrambi i filtri, come anche uno o più modalità di ordinamento. Qualora siano selezionati più ordinamenti, il questi verranno applicati in cascata, nell'ordine nome cognome cf/piva data\_nascita.

L'ultima feature corrisponde alla term-expansion: qualora il cliente non sia in possesso dell'informazione completa da ricercare, può scegliere di abilitare la ricerca di informazione parziale. Le tre modalità sono date dalle combinazioni di *PRE* e *POST*. Queste due keyword indicano se l'espansione dei termini debba essere eseguita a monte o a valle, o eventualmente su ambo le parti del



termine in oggetto: la ricerca parziale di 'Mar' con wildcard POST ('Mar\*') farà sì che il match avvenga su valori quali 'Mario', 'Marco', 'Marcello'; se viene spuntata PRE, invece, valori quali 'Federica' o 'Ludovica' produrranno il match sulla ricerca di 'ica' ('\*ica'). L'azione combinata delle due opzioni precedentemente descritte può essere ottenuta spuntando ambedue le checkbox.

Una volta riempito il form e accertatosi di aver inserito la query nell'apposito box di ricerca, l'utente può procedere ottenendo l'insieme dei risultati. L'endpoint costruito per l'elaborazione di una query prevede anzitutto l'indicazione del path fisso presso cui questo risulta richiamabile<sup>4</sup>, e al quale sono poi accodati gli elementi che andranno a mappare la scelta di una particolare feature di ricerca nella chiamata all'endpoint. Nell'esempio di ricerca riportato in figura, ad esempio, l'url definitivo della chiamata GET corrisponde al path assoluto poc'anzi citato concatenato con la stringa '?q=Marco&logic=AND&pagina=1&campi=nome&filtri=persona'.

Grazie al meccanismo di paginazione è possibile visualizzare, pagina dopo pagina, tutti i risultati prodotti dalla ricerca, e navigare avanti e indietro tra le pagine. Ciascun documento ottenuto viene mostrato nella sua interezza e evidenzierà in grassetto il valore, nel campo, che ha ottenuto il match con la query dell'utente. L'*highlighting* viene realizzato mediante la renderizzazione html del tag di grassetto così che, ad esempio, nel nominativo del primo documento in figura, '<b>Mario</b> Lissona' produca l'effetto di evidenziazione su 'Mario'.



Figura 4.15: Esempio di ricerca sul portale

<sup>4</sup><http://<host>:<port>/DsiWatsonSearchCrmServiceWeb/rest/crm/ricerca/clienti/>

### 4.3.2 Integrazione dell'assistente digitale

Ultimo step d'implementazione quello legato all'integrazione nella piattaforma di ricerca di un assistente digitale. All'esigenza, l'utente può scegliere di interloquire con l'assistente cliccando sull'icona del bot. Si è scelto di nascondere inizialmente il chatbot per renderlo il meno invasivo possibile, ed utilizzabile dall'utente solo ed esclusivamente in caso di necessità.

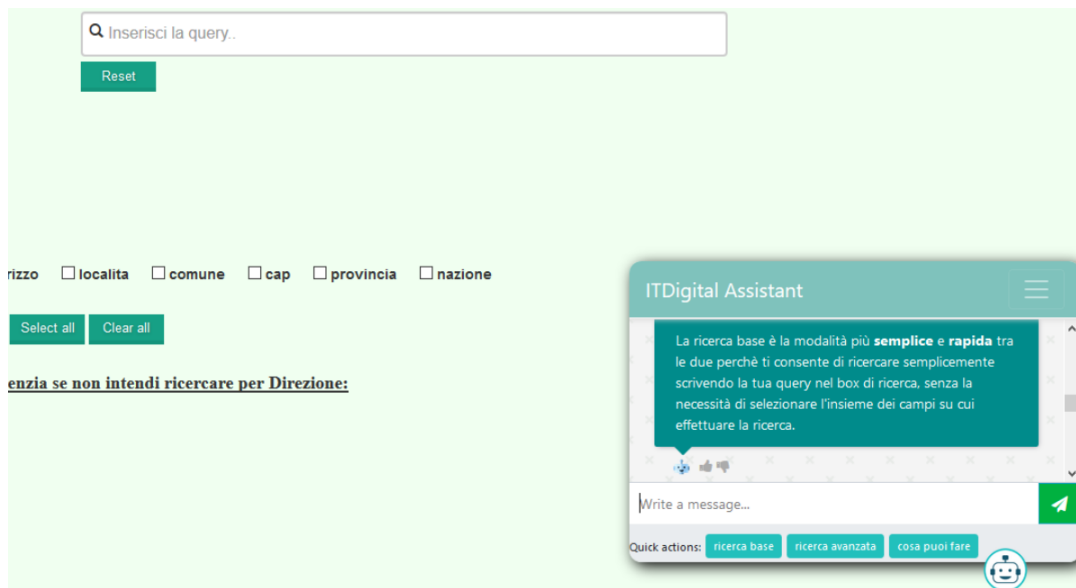


Figura 4.16: Assistente digitale a supporto dell'utente nella ricerca

Il compito del chatbot è quello di supportare l'utente nell'attività di ricerca sul portale, ed è stato pensato per riuscire a rispondere alle domande più comuni (FAQ). Il bot, opportunamente addestrato, è in grado di comprendere le richieste espresse dall'utente in linguaggio naturale e rispondere, coerentemente, in accordo alla richiesta pervenutagli. L'utente può interagire con il chatbot formulando liberamente la sua richiesta o sfruttando una serie di *quick actions* messe a disposizione, in maniera tale da velocizzare l'elaborazione della richiesta stessa, poiché espressa in maniera concisa ed esaustiva. L'assistente è in grado di rispondere per ciò che riguarda l'interazione utente con il portale di ricerca, e quindi relativamente alle funzionalità offerte dalla piattaforma, alle modalità di ricerca disponibili, oltre che a tutta una serie di domande che l'utente può avere la necessità di porre perché magari si trova in difficoltà e/o è alla sua prima interazione con il portale.

Il modello alla base del chatbot è stato sviluppato servendosi del servizio Conversation offerto da IBM Bluemix. Tale modello è abbastanza semplice perché il chatbot non riveste un ruolo attivo ma di guida in un'attività. Il flusso di dialogo che caratterizza il modello è lineare. I concetti che sono

stati modellati per consentirne il funzionamento sono quelli di *intento* ed *entità*. Mentre l'intento rappresenta ciò che l'utente vuole ottenere, l'entità è un qualcosa che aiuta il chatbot a migliorare la sua risposta. È stato definito un intento per ciascuna delle possibilità secondo cui l'utente possa porsi nei confronti del chatbot: al riconoscimento dell'intento nell'input dell'utente, il chatbot risponderà secondo quanto specificato nel flusso di dialogo.

<b>Intenti</b>	abilità, affermazione, arrivederci, darePriorità, errore, features, filtrare, grazie, info_assistant, negazione, ordinare, ricercaParziale, ricercaSegmento, saluto, tipoRicerca
<b>Entità</b>	feature (filtro, ordinamento, parziale, priorità), ricerca (ricerca base, ricerca avanzata), segmento (agenzia, direzione)

Tabella 4.3: Intenti ed entità definiti nel modello del Conversation

Per ogni intento si è fornito un insieme di frasi di addestramento modellate sulla base di come si è ritenuto l'utente finale possa esprimersi nel dialogo con l'assistente digitale. Di norma vengono fornite almeno cinque frasi per intento, di modo che la classificazione dell'input possa avvenire nel modo più corretto possibile. La scelta delle frasi su cui poi avviene l'auto-addestramento del chatbot è un'attività cruciale nella definizione del modello del Conversation e che richiede, quindi, una particolare attenzione. L'ideale sarebbe richiedere a chi di dovere, qualora il chatbot sia sviluppato per conto di un cliente, un insieme di contenuti dai quali risulta possibile estrarre ciò di cui si necessita per l'addestramento. Se il chatbot è stato pensato per sostituire l'attività umana di customer care, la strada che può essere seguita (e che è stata effettivamente seguita nel lavoro di tesi) coincide con la raccolta di un insieme di frasi caratterizzante l'interazione con l'utente in quest'attività.

Per quanto riguarda le entità, invece, ne sono state definite tre diverse: feature, ricerca, e segmento. L'entità, qualora rintracciata dal chatbot nella frase di un utente, viene adoperata per indirizzare il flusso della conversazione nel senso corretto: ad esempio, se l'utente chiede informazioni sulle tipologie di ricerca disponibili, verrà riconosciuto l'intento 'tipoRicerca' (#tipoRicerca) e verrà data una breve descrizione delle funzionalità; a questo punto, il chatbot chiede all'utente se intende avere una descrizione nel dettaglio di una tipologia in particolare, ed è qui che, a seconda del valore assunto dall'entità di ricerca (@ricerca) immessa dall'utente, si attiverà il nodo del dialogo giusto e il chatbot darà la risposta che l'utente si aspetta.

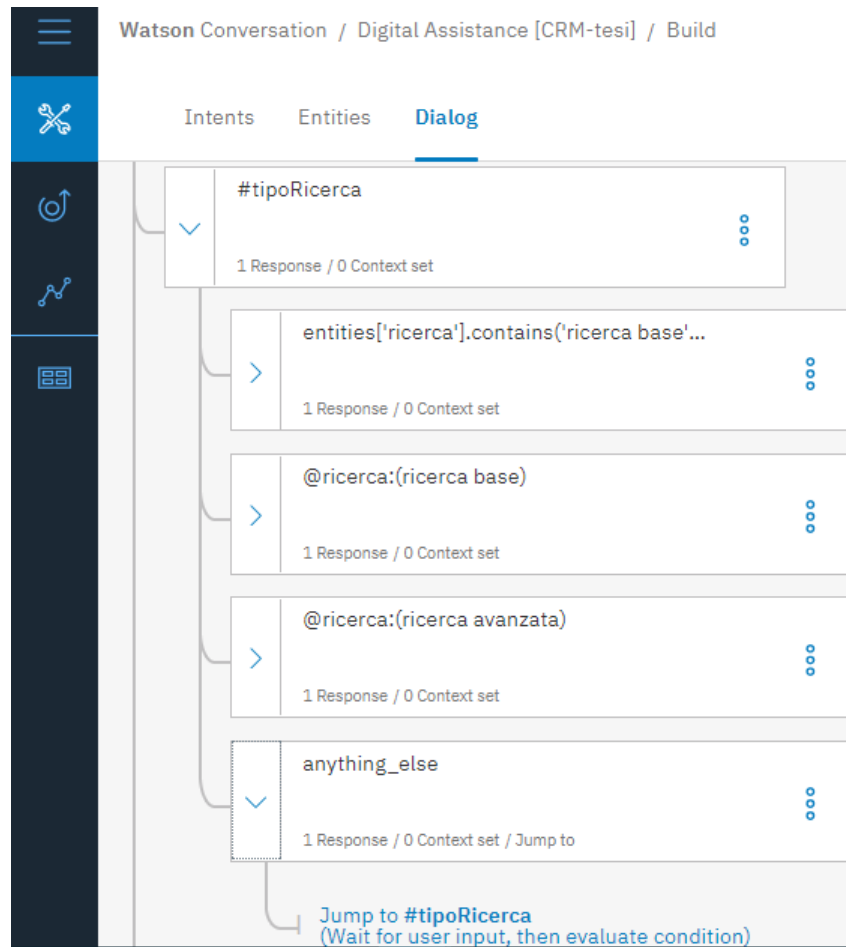


Figura 4.17: Parte del flusso di dialogo definito nel Conversation

**Integrazione dei servizi mediante Node-Red** Una volta definito il modello, questo è stato integrato nella progettazione di un'interfaccia, realizzata in AngularJS, che sfrutta tutta una serie di servizi definiti in un flusso Node-Red. Sfruttando l'approccio, tipico in Node-Red, di *flow-based programming*, si è realizzata un'applicazione modellata nel flusso tramite un insieme di blocchi. Mentre alcuni blocchi sono stati presi ed utilizzati come 'black box' (in quanto se ne conosce il tipo di elaborazione eseguita ma non necessariamente la corrispondente implementazione), altri blocchi sono stati costruiti ad hoc per la funzione che si vuole essi eseguano. Il flusso definito non è da considerare alla stregua di una sequenza di istruzioni, ma piuttosto come un insieme di flussi di dati che vengono scambiati tra i vari blocchi in maniera del tutto asincrona.

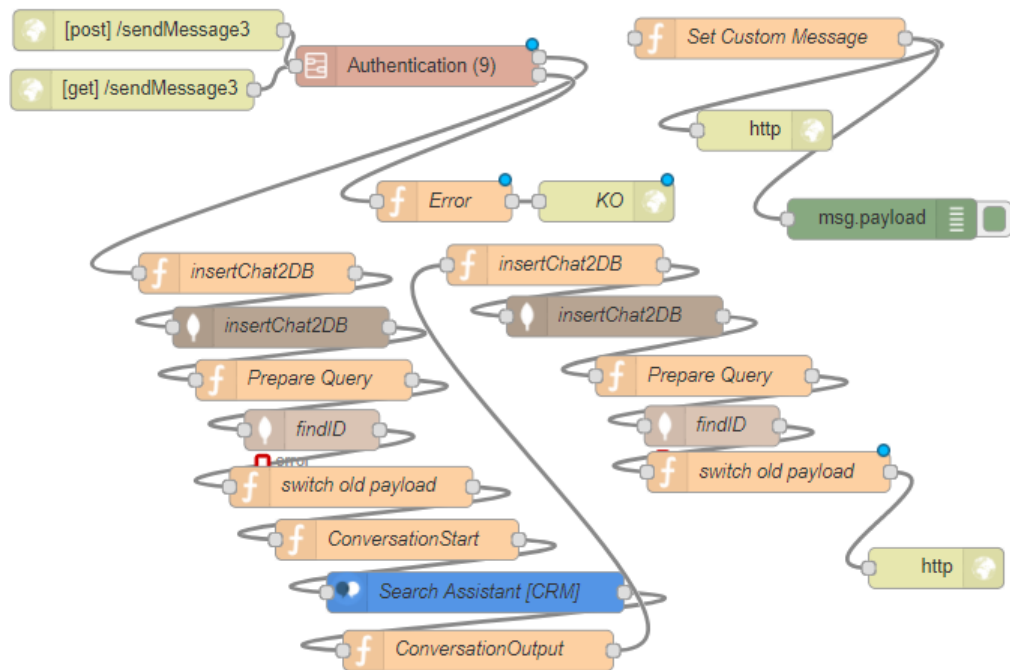


Figura 4.18: Flusso Node-Red utilizzato per la gestione del chatbot

Il frontend realizzato per l'interfaccia in stile chat si occupa di prelevare l'input dell'utente e chiamare un servizio che è quello in figura. Il nodo che fa utilizzo del modello del Conversation è il nodo 'Search Assistant [CRM]', nel quale è stata inserita, tra le altre cose, la configurazione di accesso al tool di Bluemix in termini di username, password, indicazione del *workspace ID* (spazio di lavoro ove definito il modello), ed endpoint del servizio Bluemix da richiamare. Nel flusso, a monte dello scambio di messaggi tra utente e chatbot, è stato predisposto un nodo per l'autenticazione, anche se in realtà non viene fatto uso di tale meccanismo. Più che nel chatbot, l'idea potrebbe essere quella di autenticare l'utente ancor prima, ovvero all'atto della visita del portale di ricerca, in modo da differenziare così le tipologie di utenti e limitare, ad esempio, la ricerca per segmento di agenzia solo ed esclusivamente ad una fetta di utenti. Quanto appena detto costituirà uno dei primi spunti, se non il più interessante, discusso nel capitolo finale relativo agli sviluppi futuri del progetto.

I nodi del flusso etichettati con un simbolo rappresentante una 'f' sono chiamati nodi *funzione*, e all'interno è possibile inserire, grazie ad un 'piccolo editor' del codice JavaScript che effettui l'elaborazione di cui si ha bisogno. Questa tipologia di nodi viene usata, nel flusso definito, in misura maggiore per gestire la parte legata alla *chat history*. Questa è una funzionalità che consente di tenere traccia dei messaggi scambiati nella chat, e viene realizzata mediante un meccanismo di salvataggio/recupero dei messaggi in un data-

base. Tutti i messaggi vengono memorizzati sul db assieme ad una serie di informazioni caratterizzanti il messaggio stesso, ovvero l'indicazione sul fatto che il messaggio sia dell'utente o del bot, il timestamp, gli intenti e le entità riconosciute nel messaggio (qualora questo sia dell'utente).

Servizio	Input	Output
<b>chatHistory</b>	<ul style="list-style-type: none"> <li>- chatBotID: id del chatbot,</li> <li>- qty [Opzionale]: numero di messaggi che si richiedono,</li> <li>- messageID [Opzionale]: id del messaggio per cui si richiedono i messaggi precedenti</li> </ul>	<ul style="list-style-type: none"> <li>- _id: id del messaggio,</li> <li>- timestamp: momento in cui il messaggio è stato inviato,</li> <li>- message: corpo del messaggio,</li> <li>- isResponse: 0 se è U2W (utente), 1 se è W2U (bot)</li> </ul>

Tabella 4.4: Servizio di chatHistory per tenere traccia di una conversazione

Grazie a questo meccanismo è possibile tenere in memoria il flusso della conversazione: i nodi nel flusso Node-Red etichettati con un simbolo rappresentante una foglia sono i nodi che eseguono l'operazione di lettura/scrittura sul database. L'input indicato nella tabella rappresenta ciò che l'interfaccia utente richiede al servizio, ossia una precisa quantità di messaggi a partire dal primo visualizzato in ordine di timestamp, in modo da ricreare, all'apertura del widget di chat e allo scrolling all'indietro della chat stessa, la storia della conversazione tra utente e bot. I due parametri di *qty* e *messageID* sono opzionali, in quanto:

- qualora non specificato il primo dei due parametri (*qty*) ma presente il *messageID*, viene richiesta una quantità predefinita (5) di messaggi a partire dal messaggio indicato
- se non presente l'indicazione dell'ID del messaggio a partire dal quale ottenere lo 'storico', viene richiesto il numero di messaggi scelto, qualora presente *qty*, o predefinito (10) in caso contrario

Relativamente alla risposta restituita da Node-Red (output), tra le diverse informazioni fornite, il parametro *isResponse* permette di distinguere se il messaggio è del chatbot o dell'utente, in modo da costruire correttamente la sequenza dei messaggi scambiati.

## Capitolo 5

### Conclusione e Sviluppi futuri

Completata la discussione del sistema, avendone descritto nel dettaglio ciascuna delle componenti e il ruolo da queste rivestito nell'architettura, nel capitolo finale verranno tirate le conclusioni sul lavoro di tesi, accompagnate dalle opportune considerazioni in termini di risultati raggiunti e migliorie apportabili alla soluzione.

Il presente capitolo, dunque, dopo un breve sunto di quanto emerso nei capitoli precedenti, prevede che per ciascuno dei tre macroblocchi costituenti il sistema (indexing, search layer, search-engine ui) siano presentati degli spunti di lavoro ritenuti interessanti e meritevoli di considerazione e approfondimento, con il fine ultimo di fornire ulteriore valore aggiunto al progetto.

Il modello dei dati su cui è stata costruita la soluzione è relativo ad un contesto assicurativo. Si è scelto di modellare il concetto di documento-vista associato ad un cliente in questo contesto, fornendone un insieme per ciascun cliente registrato presso la Compagnia assicurativa d'interesse. L'indicizzazione ad hoc dei dati a partire da una fonte strutturata quale un database CRM è stata quindi pensata per fornire una visione d'insieme delle informazioni caratterizzanti il particolare cliente, e consentire una ricerca full-text di queste informazioni, come se la ricerca stessa avvenisse su di un documento testuale più che su un record con una struttura rigida come quella imposta da un modello relazionale dei dati. Mediante la predisposizione di un motore di ricerca si è resa fruibile da parte dell'utente finale che interagisce con la piattaforma tutta una serie di funzionalità nell'attività di ricerca. Viene così arricchita la modalità d'interazione dell'utente con il sistema finale, anche grazie alla possibilità di servirsi del supporto di un assistente digitale.

## 5.1 Rivisitazione del modello dei dati e delega dell'attività di ricerca all'assistente digitale

**Estensione dei contenuti** Le entità secondarie a partire dalle quali viene originato un documento associato alla vista di un cliente (anagrafica o di segmento d'agenzia) sono le entità indirizzo e portafoglio. Una prima evoluzione della soluzione potrebbe prendere in considerazione l'insieme completo delle entità presenti nella base di dati, riuscendo in questo modo a ricostruire una visione del cliente completa in tutte le sue parti (veicoli, recapiti, etc.). La soluzione realizzata nel lavoro di tesi si presenta facilmente estendibile in tal senso.

**Ridefinizione del concetto di Direzione** Nella costruzione dei documenti relativi al cliente è stato utilizzato il concetto 'fittizio' di *Direzione* nell'etichettare un documento come rappresentante i soli dati dell'anagrafica del cliente. Nel dettaglio, ai campi 'Compagnia' e 'Agenzia' è stato pensato di assegnare la sigla 'D', di modo che nell'attività di ricerca la non specifica dei valori dei due campi si tramutasse in una ricerca nell'insieme dei documenti contrassegnati come di Direzione. Il concetto di Direzione può essere rivisitato e gli può essere assegnato un significato differente: più che un valore fittizio, 'D' potrebbe indicare una sorta di privilegio posseduto o meno dall'utente che utilizza il servizio di ricerca sul portale. In altri termini, agli utenti di Direzione potrebbe essere assegnato un privilegio maggiore nel senso di conferire a questi la piena visione sui clienti e, quindi, sull'insieme dei segmenti di agenzia della Compagnia Assicurativa. Questo è un concetto che trova facile applicazione, d'altronde, nel meccanismo di autenticazione del quale si parlerà nel seguito del capitolo.

**Configurazione ed utilizzo degli shards BigIndex** Relativamente ai blocchi di indicizzazione e di ricerca dei dati, inoltre, pensando la soluzione come tipica di un ambiente enterprise, potrebbe essere aggiunta scalabilità al sistema sfruttando a dovere lo strumento BigIndex API, che rappresenta la scelta ideale quando ci si trova a lavorare con archivi di database estremamente grandi. Al crescere del volume dei dati in gioco, un approccio adottabile per gestire al meglio l'elevata mole dei documenti indicizzati e soddisfare i requisiti di *high availability* e *load balancing* sarebbe quella di servirsi dei benefici ottenibili dall'utilizzo degli shards. Come descritto nel capitolo III, uno shard in BigIndex



altro non è che il risultato di un partizionamento degli indici in segmenti facilmente distribuibili e replicabili su di un insieme di istanze server di Watson Explorer Engine. BigIndex API nasconde all'utilizzatore la complessità che si cela dietro alla gestione dei segmenti e delle repliche, e si presenta quindi come uno strumento ready-to-use. Facendo uso degli shards, l'insieme dei documenti indicizzati non sarebbe più contenuto in un'unica grande search-collection ma sarebbe distribuito tra più collezioni potenzialmente su server diversi. L'indicazione sugli shards viene inserita nell'xml di configurazione dell'entità, ed è poi compito di BigIndex distribuire e replicare i segmenti di indice sulla base dei parametri scelti, quali dimensione di uno shard o numero di repliche per documento indicizzato. Relativamente alla ricerca, ci si può servire di una *bundle source* definibile sul tool di amministrazione di WEX, che rappresenta un contenitore di source: anziché indicare la singola source associata alla search collection, vengono indicate nella bundle l'insieme delle source associate alle collezioni create, di modo che l'attività di ricerca insista correttamente sull'insieme completo dei documenti.

**Completa integrazione dell'attività di ricerca nel chatbot** Il motore di ricerca e l'assistente digitale sono i due attori principali nello scenario finale della piattaforma di ricerca assistita presentato all'utilizzatore del sistema. Il primo si occupa della parte vera e propria di ricerca dei documenti mentre il secondo riveste un ruolo passivo ed esclusivamente di supporto. La scelta di tenere separate le due parti è stata una scelta dettata dal contesto, in quanto si è ritenuto più naturale predisporre una modalità di ricerca in stile Google-Like e accompagnata dalla compilazione di un form, piuttosto che delegare il tutto ad un agente conversazionale. Nulla vieta, però, in uno sviluppo futuro del lavoro, di unificare le due componenti e fare in modo che l'assistente digitale partecipi attivamente all'attività di ricerca. Tale integrazione potrebbe ottenersi ridefinendo il comportamento dell'assistente digitale: alla tradizionale azione di supporto (customer care), viene accostata la capacità di prendersi carico di una query di ricerca dell'utente e fornire a quest'ultimo l'insieme dei risultati. In altri termini, il chatbot rivestirebbe anche il ruolo che attualmente è svolto dal motore di ricerca: il fine ultimo è quello di fare in modo che il chatbot riesca ad individuare la query di ricerca nell'input dell'utente e sfruttare il layer di ricerca già presente per interrogare la collezione dei documenti e ritornare il risultato atteso.

## 5.2 Integrazione di nuovi servizi nella piattaforma

### 5.2.1 Autenticazione e differenziazione degli utenti

La soluzione realizzata non mantiene in alcun modo l'informazione legata all'utente finale, indipendentemente dal fatto che questo intraprenda un'attività di ricerca o una di conversazione con l'assistente digitale. L'idea è quella di autenticare l'utente nel momento in cui viene effettuato l'accesso alla piattaforma. In questo modo, a seconda della tipologia di utente e dei privilegi da questo posseduti, si potrà o meno avere un utilizzo completo dell'insieme di funzionalità della ricerca. Dal momento che risulta possibile ricercare sia tra le informazioni condivise del cliente che tra le informazioni dettagliate legate alla relazione agenzia-cliente, una prima differenziazione potrebbe avvenire sul *contenuto informativo* ricercabile, e quindi fare in modo che l'utente di un segmento di agenzia possa esclusivamente accedere alle informazioni condivise e a quelle del solo segmento. Al contempo, si procederebbe ad abilitare alla ricerca *full-content* un utente in possesso di particolari privilegi, come può essere colui che deve avere una visione globale del cliente (utente di 'Direzione', nel senso precedentemente descritto), completa di tutte le informazioni di indirizzo e portafoglio che quel cliente possiede.

La predisposizione di un meccanismo di login, dunque, consentirebbe di mettere in atto una diversificazione del servizio offerto sulla base del particolare utente autenticatosi.

Servizio	Input	Output
login	- uid: username per cui si chiede l'autenticazione, - password: password di accesso	- response: «OK» o «KO», - token: token di identificazione dell'utente
logout	- authToken: c.s.	- response: «OK» o «Error»
userInfo	- token: c.s.	- response: «OK» o «KO», - name: nome dell'utente, - surname: cognome dell'utente, - mail: e-mail dell'utente, - profileImage: immagine del profilo dell'utente

Tabella 5.1: Differenziazione degli utenti sulla piattaforma di ricerca

Il servizio di login richiede all'utente l'inserimento di username (*uid*, che identificherà l'utente nella piattaforma) e password. Questi dati vengono passati dal frontend a Node-Red, che si occupa di verificarne la validità mediante un nodo apposito dedicato al check dei dati con le informazioni legate all'utente sul database, e restituisce come output l'indicazione sull'esito dell'operazione e un *token* di autenticazione, opportunamente calcolato sulla base dei valori

caratterizzanti l'utente (username, password) e altre informazioni quali il timestamp relativo all'operazione di autenticazione. Il token rappresenta uno strumento del quale ci si serve per individuare, in un determinato istante e in una determinata interazione con la piattaforma di ricerca, per quale tra gli utenti abilitati all'utilizzo di questa sia stata eseguita la particolare chiamata al servizio. Questo token, quindi, viene passato ad ogni interazione dell'utente con la piattaforma, in modo da gestire correttamente la multi-utenza, andando ad estrapolare, dal token inserito nell'interazione, l'indicazione sull'utente. Dello stesso token, infine, ci si potrebbe servire per gestire la sessione browser legata ad un utente e, quindi, mediante un meccanismo di cookie, invalidare opportunamente la sessione allo scadere di un timeout predefinito. Al login, ovviamente, viene fatta corrispondere la duale operazione di logout, mediante la quale l'utente comunica di voler concludere l'attività di navigazione e ricerca sulla piattaforma. A logout avvenuto, il token di sessione viene invalidato e, ad un eventuale nuovo login il meccanismo appena descritto viene reiterato.

Relativamente all'assistente virtuale e, nel dettaglio, alla gestione della conversazione del bot con un determinato utente, un'idea sarebbe quella di prevedere anche qui un concetto di 'sessione'. La sessione in una conversazione la si potrebbe pensare alla stregua di un mantenimento del contesto, ovvero potrebbe essere implementato un meccanismo secondo cui venga tenuta traccia del contesto relativo ad una data conversazione con un utente per un certo lasso di tempo, ma che poi venga azzerato allo scadere di questo intervallo. Il chatbot, come noto, mantiene traccia del contesto o, in altri termini, di un insieme di informazioni grazie alle quali viene identificato il punto esatto nel flusso di dialogo, modellato dal Conversation, ove ci si ritrova in un dato istante. È ragionevole, dunque, pensare di resettare questo insieme di informazioni nel momento in cui si sia rilevato che un utente non abbia interagito con l'assistente digitale per un tempo maggiore di un tempo predefinito. Una prolungata mancanza d'interazione viene quindi a coincidere con un'operazione di opportuna notifica al Conversation di modo che l'utente, con un nuovo messaggio, possa interagire con l'agente come se la conversazione fosse appena iniziata (contesto pulito). L'implementazione potrebbe essere realizzata mediante la predisposizione di un job che, con una certa frequenza, vada a confrontare il timestamp attuale con il timestamp relativo all'ultimo messaggio di ciascuno degli utenti abilitati all'interazione con il chatbot e, qualora maggiore di una certa soglia, procedere con l'azzeramento del contesto.

Si noti la differenza tra i due concetti di sessione browser e di mantenimento

del contesto di conversazione: mentre la prima è legata all'interazione dell'utente con la web application nel suo insieme, il secondo è relativo esclusivamente alla conversazione utente-chatbot, e ne deriva che la distruzione di una sessione browser non implica in alcun modo il reset del contesto di conversazione (i due concetti, simili dato che poggiano entrambi sull'elemento del tempo, viaggiano però su binari paralleli).

Il servizio *userInfo*, infine, viene predisposto al fine di ottenere l'insieme delle informazioni legate all'utente attivo in un determinato istante, ovvero informazioni quali nome, cognome, e-mail e immagine del profilo. L'attività di recupero di queste informazioni viene svolta da un ulteriore nodo Node-Red definito nel flusso, similmente a quanto avviene per la gestione dell'autenticazione.

### 5.2.2 Continua evoluzione dell'attività di assistenza

**Feedback utente** Al fine di migliorare la *user experience* nell'attività di assistenza alla ricerca fornita dal chatbot, una strada che è stata intrapresa, seppure non in modo completo, prevede il coinvolgimento dell'utente finale in un processo di continua evoluzione del modello del Conversation presente alla base. Da ciò ne deriva l'inserimento del concetto di dinamicità nel modello, raggiungibile senza alcun dubbio facendo uso dei *feedback* impostabili dall'utente sul singolo messaggio ricevuto dall'agente conversazionale o, ancor meglio, sull'esperienza completa d'interazione. Il servizio descritto è già implementato nella soluzione attuale, ma con un fine diverso: una tabella sul database raccoglie le informazioni legate ai feedback ma queste poi non vengono effettivamente utilizzate ai fini illustrati poc'anzi, ma semplicemente per tenere traccia dell'attività dell'utente.

Servizio	Input	Output
<b>setFeedback</b>	<ul style="list-style-type: none"> <li>- token: c.s.,</li> <li>- messageId: id del messaggio da impostare,</li> <li>- feedback: 0 (per like) o 1 (per unlike),</li> <li>- feedback_comment [opzionale]: commento lasciato dall'utente</li> </ul>	<ul style="list-style-type: none"> <li>- response: «OK» o «KO»</li> </ul>

Tabella 5.2: Rilascio del feedback utente per l'assistente digitale

Il servizio *setFeedback* disposto su Node-Red prevede che siano passati come parametri, oltre al token identificativo dell'utente, l'id del messaggio al quale è stato associato il feedback, un booleano che indica il *like* (0) o l'*unlike*

(1), e un messaggio di testo (opzionale) contenente il commento lasciato dall'utente. L'opzionalità del parametro *feedback\_comment* sta ad indicare che sul database saranno inseriti dei record che conterranno obbligatoriamente l'identificativo del messaggio e l'indicazione di like/unlike, mentre il campo relativo al commento testuale sarà riempito all'occorrenza, soltanto quando effettivamente disponibile, e sarà di maggiore aiuto nell'attività di miglioramento del modello perché più esplicativo di un semplice like o unlike.

**Attività di monitoring** Oltre che tramite manipolazione dell'esplicito feedback rilasciato dall'utente finale, il valore aggiunto della dinamicità può essere apportato, nel modello del Conversation, mediante un meccanismo di continuo monitoraggio della conversazione. Il *live monitoring* di una conversazione consentirebbe di avere un riscontro costante sul comportamento dell'assistente digitale nell'interazione con un utente. Nel dettaglio, ci si può servire delle informazioni ottenibili come corpus della risposta, fornita dal Conversation, a seguito dell'invio di un messaggio al chatbot.

Name	Description
input <a href="#">MessageInput</a>	The user input from the request.
intents <a href="#">RuntimeIntent[ ]</a>	An array of intents recognized in the user input, sorted in descending order of confidence. Returns an empty array if no intents are recognized.
entities <a href="#">RuntimeEntity[ ]</a>	An array of entities identified in the user input. Returns an empty array if no entities are identified.
alternate_intents boolean	Whether to return more than one intent. Included in the response only when sent with the request.
context <a href="#">Context</a>	State information for the conversation.
output <a href="#">OutputData</a>	Output from the dialog, including the response to the user, the nodes that were triggered, and log messages.

Figura 5.1: Struttura Json di risposta alla API Conversation 'Send message'

Grazie alle informazioni relative agli intenti e alle entità rilevate in un messaggio dell'utente (rispettivamente, gli array *intents* ed *entities*), si potrebbe pensare di andare a correggere eventuali anomalie o comportamenti sbagliati tenuti dal chatbot: negli array indicati è reso disponibile l'elenco di intenti e di entità con associata la relativa confidenza di classificazione, e tale misura potrebbe essere utilizzata al fine di prendere le dovute decisioni, a seconda

dello scopo che si vuole perseguire. In ogni caso, dato che sul database viene mantenuta traccia dei messaggi scambiati tra utente e bot e, in particolare, ad essere memorizzato in un record non è il semplice messaggio di testo ma l'insieme di informazioni ottenute dalla risposta (vedi figura), tale meccanismo potrebbe essere esteso ad un'attività di *analysis* ritardata nel tempo. Il monitoraggio in tempo reale potrebbe essere invece sfruttato per correggere istantaneamente il comportamento del bot e, ad esempio, prevedere un'azione diversa dalla tradizionale risposta automatizzata (come ad esempio la possibilità di esser messi in contatto con un operatore umano) qualora ci si accorga che i valori della confidenza siano al di sotto di una certa soglia (indice di un potenziale errore di misclassificazione dell'input dell'utente).

Numerosi fattori possono influenzare l'esperienza che un utente vive utilizzando un sistema. Il numero delle variabili potenzialmente in gioco non è di certo limitato, ragion per cui le attività di analisi, in tempo reale e non che siano, costituiscono sicuramente uno strumento tramite il quale potersi continuamente migliorare e fornire quindi all'utente una user experience sempre più completa e soddisfacente.

# Bibliografia

- [1] C. Manning, *An Introduction to Information Retrieval*. New York, Cambridge University Press, 2008.
- [2] G. Sacheli, "Appunti di information retrieval," 2017. Available at <https://www.evemilano.com/appunti-information-retrieval/>.
- [3] R. Faramundo, "Sistemi per il recupero delle informazioni, information retrieval," 2016. Available at <http://slideplayer.it/slide/3646565/>.
- [4] R. Ridi, "Nozioni di information retrieval," 2007. Available at <http://lettere2.unive.it/ridi/info-retr.pdf>.
- [5] S. M. Beitzel, *On Understanding and Classifying Web Queries*. PhD thesis, Graduate College of the Illinois Institute of Technology, Chicago, 2006.
- [6] A. Albano, "Sistemi per l'archiviazione e recupero delle informazioni," 1999. Available at <http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/bdd-infuma/albano.pdf>.
- [7] P. Maresca, "Introduzione ad apache lucene," 2009. Available at <http://www.html.it/articoli/introduzione-ad-apache-lucene-1/>.
- [8] A. Sonawane, "Using apache lucene to search text," 2009. Available at <https://www.ibm.com/developerworks/library/os-apache-lucenesearch/>.
- [9] F. Scioscia, "Information retrieval con apache lucene," 2014. Available at <http://www.inginfpoliba.eu/>.
- [10] GruppoUnipol, "Intelligenza artificiale e robotica," 2017. Available at [http://www.unipol.it/sites/corporate/files/documents/quaderno\\_intelligenza-artificiale-e-robotica\\_2017.pdf](http://www.unipol.it/sites/corporate/files/documents/quaderno_intelligenza-artificiale-e-robotica_2017.pdf).
- [11] J. Mugan, "Chatbots: Theory and practice," 2017. Available at <https://www.linkedin.com/pulse/chatbots-theory-practice-jonathan-mugan>.

- [12] A. Geitgey, "Machine learning is fun part 5: Language translation with deep learning and the magic of sequences," 2016. Available at <https://medium.com/@ageitgey/>.
- [13] O. Vinyals and Q. Le, "A neural conversational model," 2015.
- [14] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," 2015.
- [15] A.Lazaridou, R.Bernardi, D.T.Nguyen, and M.Baroni, "Unveiling the dreams of word embeddings: Towards language-driven image generation," 2015.
- [16] E. Pucci, "Bot economy: esiste davvero? ecco come i chatbots cambieranno la nostra economia," 2017. Available at <http://www.awhy.it/chatbot/>.
- [17] W.-J. Chen, B. Adams, C. Dean, S. S. Naganna, U. K. Nandam, and E. Thorne, *Building 360-Degree Information Applications*. ibm.com/redbooks, 2014.
- [18] IBM, "Ibm knowledge center - home of ibm product documentation." <https://www.ibm.com/support/knowledgecenter>.
- [19] *Developing Applications Using the BigIndex API (Draft)*. IBM, 2014.
- [20] IBM, "Conversation." Available at <https://console.bluemix.net/docs/services/conversation/getting-started.html#gettingstarted>.
- [21] P. Patierno, "Installare node-red." Available at <http://www.html.it/pag/51924/installare-node-red-node-js-per-liot/>.