Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

SPEAKIN' ABOUT – UNO STRUMENTO PER L'ANNOTAZIONE SEMANTICA DI TESTI

Elaborato di Laurea di:

Marco Polonio
Matricola: 663109

Matteo Rebulla
Matricola: 661877

Relatore: Prof. Marco COLOMBETTI

Correlatore: Ing. Davide EYNARD

Anno Accademico 2007-2008

INDICE

INTRODUZIONE

L'ingegneria della conoscenza e il Web Semantico

Internet contiene una miniera di informazioni immediatamente comprensibili e utilizzabili dall'uomo. Ma come fa un computer a comprendere queste informazioni e ad utilizzarle? L'ingegneria della conoscenza nasce proprio per risolvere questo problema, proponendo di organizzare queste informazioni in schemi logici strutturati, chiamati "basi di conoscenze" o "ontologie". Queste basi di conoscenze sono utilizzabili da un computer, che "ragiona" su di esse tramite le regole della logica matematica ed è in grado di dedurre conclusioni a partire dalle conoscenze contenute in esse.

Allo stato attuale, solo una piccola parte del Web è organizzata in questo modo. Nel Web concepito dal ricercatore del CERN Tim-Berners Lee nel 1989, i collegamenti fra i documenti sono creati sia manualmente, cioè dallo sviluppatore del sito, che automaticamente, ad esempio attraverso l'argomento di cui parla un documento (e cioè in maniera semantica). Questa parte relativa ai collegamenti automatici semantici è argomento di grande attualità nelle università e sta cominciando a svilupparsi proprio negli ultimi anni tramite molte nuove tecnologie (come ad esempio Protegé, Pellet, Pollo, Jena...). Una volta ultimata, sarà possibile effettuare ricerche sul Web non soltanto del significante delle parole contenute in un documento o nel suo titolo (cioè delle lettere che formano quella parola) ma anche (e soprattutto) del loro significato.

Problemi del Web Semantico

Proprio perché si trova nella fase iniziale del suo sviluppo, il Web Semantico non è ancora alla portata dell'utente medio di Internet, che non conosce le nuove tecnologie e non le utilizza. La nostra risoluzione di questo problema sta nello scrivere un software dotato di interfaccia grafica che permetta a qualunque utente del Web di utilizzare indirettamente queste tecnologie. Nello specifico, il nostro software dà la possibilità all'utente medio di dire qual è il significato di una parola contenuta in un sito e di associare ad essa un collegamento alla pagina che spiega il significato di quella parola,

o, più semplicemente, parla dell'argomento a cui essa si riferisce. I collegamenti da una pagina a un'altra quindi, non sono più creati dal solo creatore del sito, ma anche da un qualunque utente del Web, e sono recuperabili anche da un altro utente purché l'operazione venga registrata da un server.

Organizzazione dei capitoli

Il presente elaborato è diviso in sei capitoli:

Capitolo 1: Stato dell'arte (concetti teorici e progetti esistenti in quest'area);

Capitolo 2: Conoscenze preliminari (breve descrizione delle tecnologie utilizzate dalla nostra applicazione);

Capitolo 3: Analisi (casi d'uso e specifica dei requisiti del software);

Capitolo 4: Architettura (implementazione del software);

Capitolo 5: Test e Evaluation (esempi di utilizzo del software);

Capitolo 6: Conclusioni e sviluppi futuri (pregi, limiti e possibili sviluppi futuri del software).

1. STATO DELL'ARTE

1.1 La logica del primo ordine e la logica descrittiva

Il nostro problema è quello di riuscire a rappresentare le conoscenze in modo che siano leggibili dal computer. Di conseguenza, dobbiamo usare un linguaggio simbolico che il computer possa decodificare e utilizzare. Il riferimento per questo tipo di rappresentazioni è la *logica simbolica,* e in particolare la *logica dei predicati del primo ordine* (*first order logic,* FOL). In questa logica, abbiamo un insieme non vuoto chiamato *universo* che contiene tutti gli individui che vogliamo rappresentare. Di questi individui possiamo rappresentare le *proprietà* o *relazioni* che intercorrono fra loro. Queste relazioni possono essere di ordine 1, e in questo caso vengono chiamate predicati e sono relative a un solo individuo, oppure di ordine 2 o più, e in questo caso legano più individui. Ad esempio, la scrittura PADRE(x) indica che l'individuo x è padre, e che quindi PADRE è il predicato di x, mentre invece la scrittura COMPAGNI_DI_CLASSE(x,y) indica che gli individui x e y sono compagni di classe, ed è quindi una relazione fra x e y.

In FOL possiamo usare i seguenti simboli:

$\forall$: quantificatore universale, "per ogni"

$\exists$: quantificatore esistenziale, "esiste"

$\rightarrow$: implicazione, "se... allora... "

$\leftrightarrow$: coimplicazione, "se e solo se... allora..."

$\wedge$: congiunzione, and

$\vee$: disgiunzione inclusiva, or

$\neg$: negazione, not

I quantificatori servono a dichiarare il numero di individui che hanno una certa proprietà. Ad esempio la proposizione $\forall$x(PERSONA(x)) significa "tutti gli individui dell'insieme universo sono persone", mentre $\exists$x(PERSONA(x)) significa "nell'insieme

universo esiste almeno una persona". Con gli altri simboli logici possiamo definire relazioni più complesse, come ad esempio ∀x∀y((CLASSE(x)=CLASSE(y))↔(COMPAGNO_DI_CLASSE(x,y))), che significa "gli individui x e y sono compagni di classe se e solo se stanno nella stessa classe".

La logica FOL è molto espressiva ma non può essere utilizzata per un ragionamento automatico da parte di un computer perché non è decidibile. I concetti di *decidibilità* e *indecidibilità* sono legati agli studi dell'Informatica Teorica, che studia le capacità massime di eseguire un algoritmo da parte di un computer. Da questi studi si vede che per alcuni algoritmi il computer potrebbe non arrivare alla fine dell'algoritmo stesso e quindi finire in stallo e non fornire una risposta finale. Uno di questi è l'algoritmo di ragionamento automatico per fornire una risposta vera o falsa a partire da premesse in FOL. Per questo motivo, l'ingegneria della conoscenza si basa su una logica meno espressiva ma decidibile, chiamata *logica descrittiva* (*description logic,* DL). In essa troviamo tutti i simboli di FOL tranne i quantificatori.

1.2 Cos'è un'ontologia

Un'ontologia, anche detta "base di conoscenze", è una descrizione di una certa area del sapere tramite asserzioni in DL. Essa è divisa in due parti, chiamate TBOX (terminological box) e ABOX (assertion box). La TBOX contiene le conoscenze generali che riguardano quell'area del sapere, mentre la ABOX contiene le conoscenze particolari. In un'ontologia riguardante il cinema, ad esempio, la TBOX potrebbe contenere frasi come "Un regista è una persona" oppure "Un regista ha diretto almeno un film", mentre la ABOX potrebbe contenere "Fellini ha diretto il film *La dolce vita*" oppure "Il film *La dolce vita* è stato girato nel 1960". In linguaggio formale, questa ontologia verrebbe scritta nel seguente modo:

TBOX:

1. REGISTA ⊆ PERSONA

2. HaDiretto: PERSONA → FILM

3. ÈStatoGiratoNel: FILM → INTEGER

4. REGISTA ⊆ ∃HaDiretto

5. FILM ⊆ =1 ÈStatoGiratoNel

ABOX:

6. REGISTA(Fellini)

7. FILM(LaDolceVita)

8. HaDiretto(Fellini, LaDolceVita)

9. ÈStatoGiratoNel(LaDolceVita, 1960)

Le parole PERSONA, REGISTA e FILM si chiamano *classi* o *concetti*, e sono insiemi che contengono singole istanze di PERSONA, REGISTA e FILM. Queste istanze vengono chiamate *individui*. Nel nostro esempio gli *individui* sono Fellini e LaDolceVita. Fellini è un'istanza di REGISTA e conseguentemente di PERSONA, mentre LaDolceVita è un'istanza di FILM (asserzioni 6 e 7). Le altre parole che troviamo, cioè HaDiretto e ÈStatoGiratoNel, sono le relazioni che sussistono fra le classi, e vengono chiamate *ruoli*. Sono definiti nella TBOX come "classe1 → classe2". Classe1 viene chiamata dominio, ed è cioè l'insieme da cui il ruolo parte, mentre classe2 è detta codominio, ed è l'insieme al quale il ruolo arriva. Nel nostro esempio, il ruolo HaDiretto va dall'insieme delle persone a quello dei film, mentre il ruolo ÈStatoGiratoNel va dall'insieme dei film a quello degli interi, che è un insieme predefinito della logica DL (asserzioni 2 e 3). I ruoli vengono usati in seguito per definire le classi, attraverso le caratteristiche peculiari che le distinguono dalle altre classi. Nel nostro esempio, una persona per essere considerata regista deve avere diretto almeno un film, e questo viene dichiarato nell'asserzione numero 4. Allo stesso modo, un individuo nell'ontologia, per poter essere considerato un film, deve essere stato diretto da qualcuno e deve essere stato girato in un certo anno (asserzioni 4 e 5). Infine,

i ruoli che vediamo nell'ABOX collegano i singoli individui fra di loro. Nelle asserzioni 8 e 9 diciamo che Fellini ha diretto La Dolce Vita, e che La Dolce Vita è stato girato nel 1960.

Ora che abbiamo definito classi, individui e ruoli, possiamo descrivere il significato dell'operatore $\subseteq$, chiamato *sussunzione*. La sintassi dell'utilizzo della sussuzione è la seguente: "classe1 $\subseteq$ classe2", e significa letteralmente "la classe1 è contenuta nella classe2". Nell'asserzione 1 diciamo letteralmente che l'insieme dei registi è contenuto nell'insieme delle persone, che in linguaggio naturale si leggerebbe "i registi sono persone".

1.3 Annotazioni

Un'annotazione, nel linguaggio dell'informatica, è un commento aggiunto a una specifica parte di un documento (character-level annotation) o al documento stesso (document-level annotation). Le annotazioni possono essere manuali, automatiche o semi-automatiche.

Annotazione manuale: Annotazione aggiunta da un utente. Un esempio di questo tipo di annotazioni è dato dai commenti al codice sorgente di un programma, che vengono utilizzati in informatica per descrivere brevemente le operazioni che compie una certa porzione di codice.

Annotazione automatica: Annotazione aggiunta dal computer senza consultazione dell'utente.

Annotazione semi-automatica: Annotazione suggerita dal computer che ha bisogno della conferma dell'utente per essere aggiunta.

Oltre a questa distinzione, è anche possibile definire "semantiche" le annotazioni che descrivono il significato di una parte di testo o che sono esse stesse descritte tramite semantica (come nel caso di Annotea, vedi sezione 1.4).

Il nostro sistema fa utilizzo di annotazioni semantiche semi-automatiche.

1.4 Sistemi di annotazioni esistenti

Al giorno d'oggi, esistono parecchi sistemi di annotazioni. Di seguito ne elencheremo alcuni con una piccola descrizione.

Annotea (http://www.w3.org/2001/Annotea/) è un server di annotazioni semantiche manuali che migliora la collaborazione tra gli utenti e il Web tramite annotazioni e bookmark basati su metadati condivisi. Utilizza uno schema RDF per descrivere le annotazioni come metadati e XPointer per il recupero delle annotazioni dal documento. Per Annotea sono stati scritti alcuni client, fra cui il più usato è Annozilla, che è un'estensione del browser Mozilla. Poiché Annotea è open-source, per esso è possibile sviluppare altri client. Un client per Annotea deve essere in grado di comprendere i metadati generati dal server e di stabilire una connessione tramite il protocollo HTTP.

Magpie (http://kmi.open.ac.uk/projects/magpie/main.html) utilizza l'infrastruttura di un'ontologia per creare annotazioni semantiche in maniera dinamica. Questo tool è stato sviluppato come estensione di un browser e mette a disposizione ulteriori informazioni sul testo annotato, che vengono recuperate solo dalla propria ontologia, e non da risorse esterne. Lo scopo principale del progetto è quello di risolvere i problemi di altri sistemi di annotazione, che secondo gli sviluppatori sono poco user-friendly e molto invasivi, in quanto modificano troppo la visione della pagina annotata.

KIM (http://www.ontotext.com/kim/index.html) è una piattaforma software per annotazioni automatiche, indicizzazioni e recupero di informazioni. L'approccio di KIM è basato sull'assunzione che le entità con un nome proprio devono essere trattate in un modo particolare perché identificano un individuo, a differenza delle entità che identificano concetti, relazioni e attributi universali. Per riconoscerle all'interno di un testo, utilizza la tecnologia NLP (Natural Language Processing), riferendola a un'ontologia predefinita.

SemTag (http://www2003.org/cdrom/papers/refereed/p831/p831-dill.html) è il sistema di annotazioni più grande che esista al momento. Fa parte del progetto di ricerca

WebFountain. I ricercatori IBM del centro Almaden, vicino San Francisco, hanno utilizzato SemTag per annotare circa 264 milioni di pagine web e generare 434 milioni di annotazioni semantiche disambiguate, che sono messe a disposizione da un server di categorizzazione come metadati. Per definire le classi di annotazioni, SemTag utilizza l'ontologia TAP, che è molto simile all'ontologia utilizzata da KIM. Per superare il problema della disambiguazione, SemTag fa uso di un modello di spazio vettoriale per associare un concetto alla classe corretta o per dire che un concetto non corrisponde a una certa classe di TAP. Per fare questo viene comparato il contesto della parola (10 parole a sinistra e 10 a destra) con i contesti degli individui in TAP che hanno degli alias compatibili con quella parola. TAP è costruita in modo da non avere troppe entità che condividono lo stesso alias, e questo rende la disambiguazione più semplice. SemTag è stato sviluppato come un'architettura parallela di cui ogni nodo annota circa 200 documenti al secondo. Dai risultati della sua esecuzione, gli autori hanno trovato che l'80% delle annotazioni risulta semanticamente corretta, ovvero che il contesto della parola annotata è stato correttamente individuato dal sistema.

2. CONOSCENZE PRELIMINARI

2.1 Conoscenze preliminari: estensione di Firefox

Per apprendere al meglio il funzionamento dell'applicazione, è bene dare prima qualche nozione di alcuni software o servizi informatici che sono stati integrati nello sviluppo del progetto.

Innanzitutto, la nostra applicazione consiste in una estensione per il browser Mozilla Firefox. Un'estensione non è altro che un plugin aggiuntivo che permette di integrare nuovi servizi nel browser. È possibile trovare molte estensioni messe on-line dagli sviluppatori al seguente indirizzo: *https://addons.mozilla.org/it/firefox/browse/type:1*.

2.2 Conoscenze preliminari: Apache Tomcat

Apache Tomcat è una applicazione Java che ci permette di lavorare con le tecnologie JavaServlet e JavaServerPages sviluppate da Sun. A noi, infatti, interessa poter far interagire il codice Javascript dell'estensione di Firefox con il codice Java utilizzato implementare i server remoti. C'è quindi bisogno di un mezzo di trasporto che sia in grado di far scambiare informazioni attraverso il web. Di questo si occupa appunto il servlet contenuto in Tomcat.

Tomcat permette l'interazione client-server tramite il protocollo HTTP. All'atto di richiesta del servizio viene creata una istanza dell'oggetto Request e, successivamente, una istanza dell'oggetto Response che rappresenta la risposta alla richiesta del client.

2.3 Conoscenze preliminari: servizi RMI

A titolo informativo, parleremo in questo paragrafo dei servizi forniti dal pacchetto RMI. Innanzitutto specifichiamo che questi servizi possono essere utilizzati solo in Java, dato che il pacchetto RMI non sono altro che delle librerie per Java. Il loro compito è quello di fornire un protocollo per lo scambio di dati tra due o più applicazioni Java che possono essere locali o anche remote. L'utilizzo di questi servizi è

molto semplice, infatti, permette di lavorare sempre ad oggetti senza il bisogno di dover specificare di inviare i singoli byte e di dover riaccorparli non appena giunti a destinazione.

Al momento non si è ancora parlato della struttura del nostro progetto, ma vedremo in seguito che RMI ci sarà utile per mettere in comunicazioni i due server che abbiamo implementato.

2.4 Conoscenze preliminari: Jena

Nel capitolo precedente si era parlato delle annotazioni semantiche che stanno praticamente alla base del nostro progetto. Come già visto, le annotazioni semantiche richiedono un certo tipo di conoscenze che vengono rappresentate sotto forma di ontologie. Ci servirà quindi un servizio che ci permetta di compiere interrogazioni sulle basi di conoscenze. Per questo scopo utilizzeremo le librerie di Jena per Java, create appositamente per il Web semantico. Queste librerie ci permetteranno di lavorare sulle ontologie, direttamente da Java senza il bisogno di avere un ulteriore programma di reasoning.

3. ANALISI

3.1 Panoramica del progetto

Prima di entrare nel dettaglio, è bene fare una panoramica del progetto mettendo in evidenza i vari attori che devono interagire tra di loro.

L'idea della nostra applicazione è quella di permettere a un qualsiasi utente di associare un determinato concetto a una o più parole selezionate in un testo; ogni concetto sarà poi, a sua volta, relazionato a un URL specifico. Ora, queste informazioni –che d'ora in poi chiameremo metadati– verrano utilizzate per creare i collegamenti automatici relativi alle parole a cui è stato associato un concetto. Ad esempio, è possibile che in una pagina web compaia il nome "Verdi" e che, nel contesto della pagina, si riferisca al compositore di musica classica Giuseppe Verdi. Bene, è possibile allora associare l'individuo "COMPOSITORE - Giuseppe Verdi" alla parola "Verdi" che si trova in quella pagina Web. In questo modo, ogni volta che un qualsiasi utente (che usa la nostra applicazione) caricherà quella determinata pagina, verrà automaticamente associato alla parola "Verdi" un link relativo a "Giuseppe Verdi" (es. http://www.giuseppeverdi.it).

Per fare ciò, si è scelto di implementare una estensione di Firefox che dovrà interagire con alcuni server che consentano la gestione delle informazioni.

3.2 Casi d'uso

Come si è già detto, oltre all'estensione per Firefox, il nostro modello architetturale prevede la presenza di servizi lato server. In particolare, ne sono serviti due: il primo –KMTool– che contiene le informazioni relative alla conoscenza, e il secondo –MetadataServer– necessario per salvare i metadati creati dagli utenti. Si è scelto di tenere separati i due server per possibili estensioni future e, comunque, per rendere più facili le operazioni di modifica o di aggiornamento. Infatti il KMTool, che si preoccupa di fornire la conoscenza ed i vari servizi di ragionamento, è del tutto indipendente dal MetadataServer, così, se dovesse essere necessaria una modifica alla gestione dei

metadati, non dovranno esserci ripercussioni anche sul KMTool. Viceversa, modifiche o estensioni apportate al KMTool non richiederanno l'implementazione di un nuovo MetadataServer. In sintesi, ognuno dei tre attori (FirefoxPlugin, KMTool, MetadataServer), ha la possibilità di essere aggiornato con future estensioni, senza che sia necessario riscrivere tutta l'applicazione.

Il seguente use-case diagram (vedi *Immagine 3.2.1*), dà una rappresentazione generale delle interazioni tra i vari attori dell'applicazione e un'idea delle varie operazioni che possono compiere.

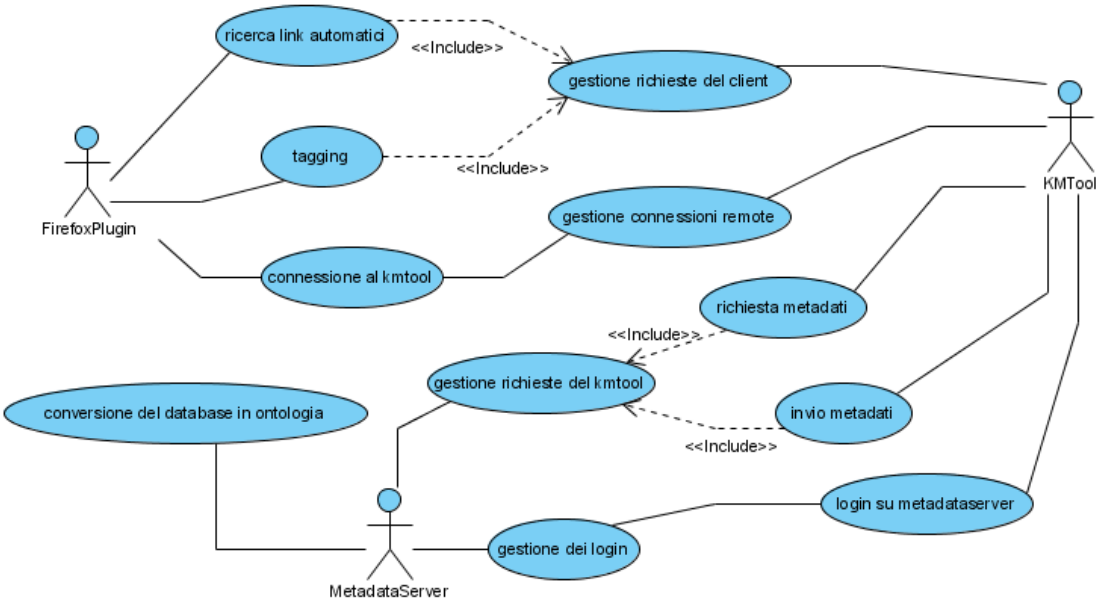


Immagine 3.2.1: *diagramma dei casi d'uso.*

Vediamo ora i ruoli principali di ogni attore, partendo dall'estensione di Firefox (o plugin) dato che è quella parte dell'applicazione con cui l'utente interagisce.

Innanzitutto, è bene specificare che il nostro plugin entra in funzione non appena il browser richiede il caricamento di una pagina ad un server HTTP. Infatti viene attivata subito la funzione di *autolinking* che, per mezzo dell'URL, richiede immediatamente al

KMTool i possibili metadati relativi a quell'indirizzo internet. I metadati sono quelle informazioni create dagli utenti che associano una stringa, o parola, ad un concetto o ad un individuo. La creazione di questi metadati è possibile grazie all'operazione di *tagging* dell'estensione di Firefox. L'utente infatti, dopo il caricamento della pagina web (e quindi anche dopo il termine dell'operazione di *autolinking*), può selezionare una parte di testo ed attivare la funzione di *tagging* del plugin; a questo punto gli sarà concesso di scegliere l'insieme generale a cui fa riferimento la parola (poi si vedrà che in pratica non è altro che la selezione di una ontologia) e conseguentemente il concetto specifico –o meglio, una istanza di un concetto che chiameremo individuo– più appropriato al testo in esame. Come si può ben intuire, al ricaricamento della medesima pagina, ora si potranno vedere quelle parole che hanno subito l'operazione di *tagging* collegate, tramite un link, ad un specifico sito internet. Ad esempio, il link che si genererà sulla parola "Bach" sarà "http://www.jsbach.org", ovvero il suo sito ufficiale (tutti questi indirizzi internet sono recuperabili dalle ontologie).

Nel paragrafo precedente si è data un'idea di ciò che sono i metadati; come è facilmente intuibile, quest'ultimi verranno salvati sul MetadataServer. Questo server, infatti, si occupa esclusivamente di archiviare i metadati durante la fase di *tagging*, e di recupero dei metadati durante la fase di *autolinking*. In entrambi i casi, il MetadataServer non dialoga direttamente con l'estensione di Firefox, ma passa attraverso il KMTool in modo da poter tenere distinta la parte di conoscenza dalla parte di dati. Si è poi anche scelto di introdurre una funzione prettamente didattica che consiste nella conversione dell'archivio dei metadati (che non è altro che un database) in una ontologia sulla quale è possibile applicare servizi di ragionamento.

Passiamo ora alle competenze del KMTool. Come si è già detto, il KMTool è un server che permette al plugin lo scambio di informazioni per le funzioni di *tagging* e di *autolinking*, e può praticamente essere visto come il fulcro dell'applicazione. Durante la fase di *autolink*, il KMTool ricopre il ruolo di interfaccia tra l'estensione di Firefox ed il MetadataServer. Il plugin, infatti, specifica l'URL della pagina web che si sta visitando al KMTool, e quest'ultimo richiede al MetadataServer gli eventuali metadati

corrispondenti all'URL precedentemente segnalato. Ricevuti i metadati, avrà poi il compito di rispedirli all'estensione di Firefox che completerà l'operazione di *autolinking*.

La fase di *tagging*, dal lato del KMTool, è invece più complicata, ma essenzialmente funziona in questo modo: per prima cosa ricerca la parte di testo selezionata, inviatagli dal plugin, nella base di conoscenze. A questo punto entrano in gioco le ontologie: il nucleo del KMTool, infatti, è composto da una serie di ontologie suddivise per generi. Ad esempio sarà possibile trovare una ontologia sulla musica classica, una sulla politica italiana, una sul cinema, una sulle automobili e via discorrendo. Ognuna di queste, poi, contiene un certo numero di concetti che a loro volta ospitano le istanze di concetto – individui– alle quali è associato un indirizzo internet che verrà utilizzato per l'operazione di *autolinking*.

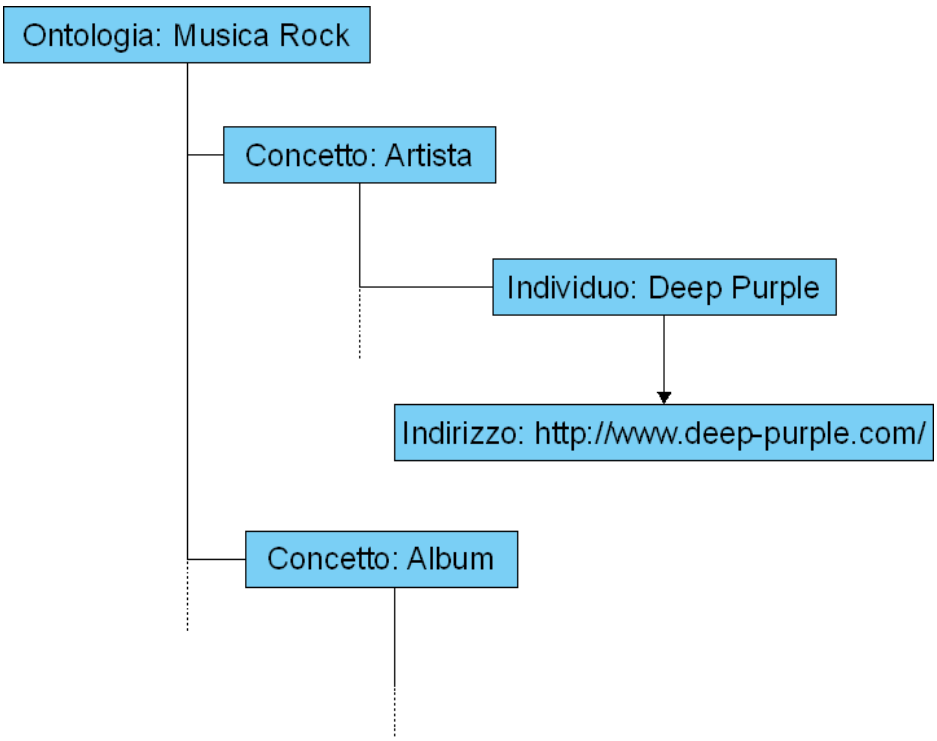


Immagine 3.2.2: *esempio dell'organizzazione di una ontologia.*

A questo punto l'utente avrà la possibilità di scegliere l'individuo che ritiene essere inerente al testo. È possibile infatti che una parola con stesso significante, abbia però un significato diverso; sta all'utente selezionare il concetto appropriato. È anche possibile, però, che nelle ontologie, per mancanza della conoscenza completa, non sia presente l'individuo adatto al contesto della pagina web. In questo caso sarà possibile comunque selezionare l'ontologia ed il concetto generico più adatti –attenzione, prima si selezionava una istanza di concetto, mentre ora si seleziona un concetto visto come oggetto astratto– e tramite un motore di ricerca, si tenterà di risalire al sito più adatto per la creazione degli autolink. Ora, le informazioni necessarie vengono recuperate dal KMTool, il quale si occupa di creare il metadato da inviare al Metadataserver.

È da specificare, però, che l'idea di suggerire il concetto o individuo tramite l'utilizzo delle ontologie è solo uno dei possibili approcci a tale problema. È possibile infatti appoggiarsi ad altri sistemi semiautomatici che ci permettono di recuperare le informazioni necessarie. Ad esempio avremmo potuto compiere interrogazioni su Freebase (http://www.freebase.com/), senza dovere necessariamente creare delle ontologie ad hoc.

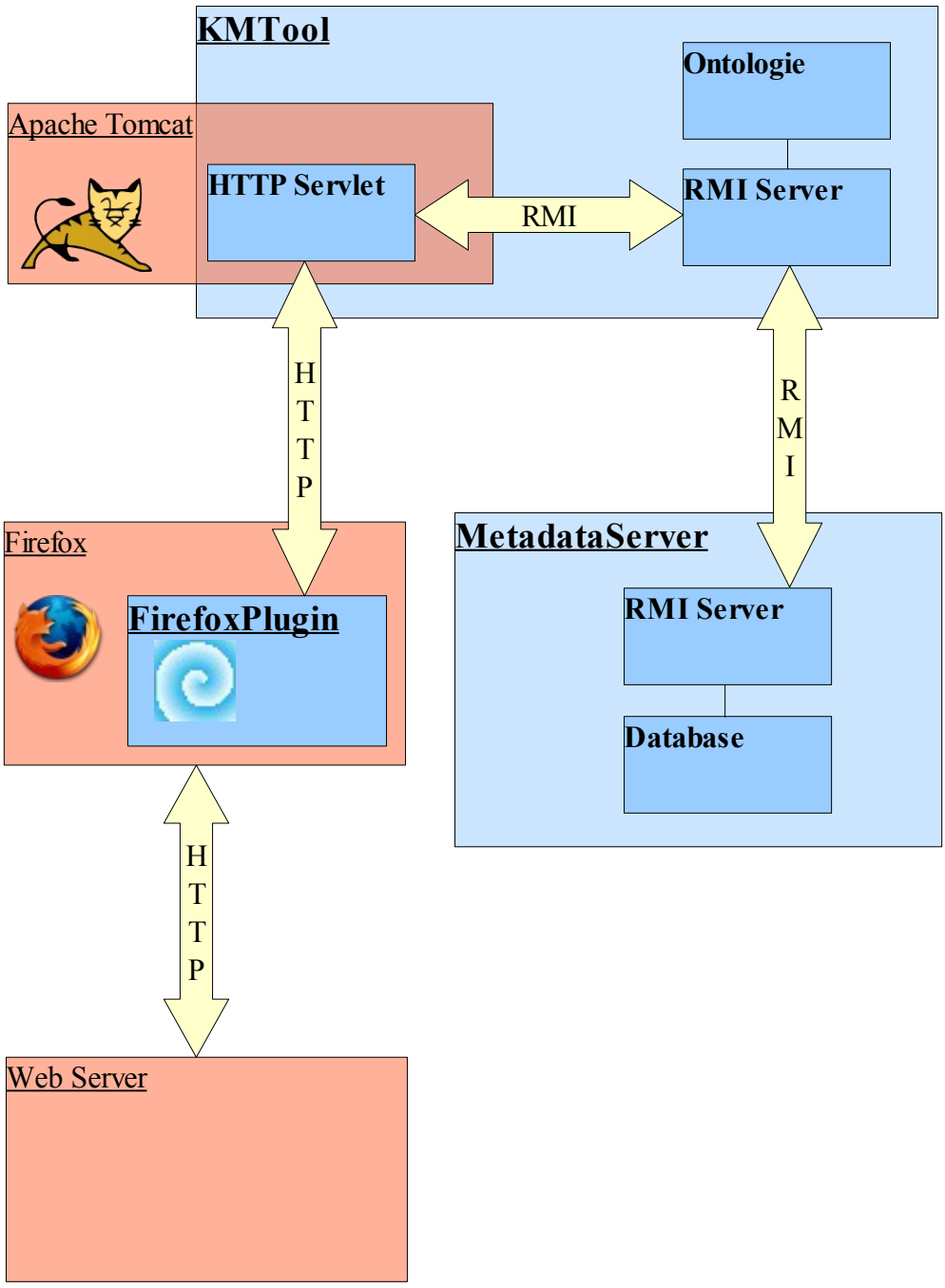
4. ARCHITETTURA

4.1 Schema generale

Speakin' About è stato sviluppato secondo il modello client-server, applicato a due server e un client. I due server sono stati sviluppati in Java, mentre il client è un'estensione di Firefox scritta in javascript.

- Server dei metadati – Ospita su database i metadati delle pagine Web visitate dagli utenti dell'estensione di Firefox. È dotato di un'interfaccia testuale con dei comandi a console per gestire il database e per creare un'ontologia a partire da esso. Di seguito verrà chiamato *MetadataServer*.
- Server di gestione della conoscenza – Ha a disposizione alcune ontologie che legge su richiesta dell'estensione di Firefox. È diviso in due parti: la prima è un server HTTP che accetta le connessioni dell'estensione di Firefox, mentre la seconda è un server RMI che si collega al MetadataServer, grazie a un'interfaccia testuale con dei comandi a console. La connessione interna fra queste due parti avviene tramite un oggetto RMI remoto contenuto all'interno del server HTTP che si connette in locale al server RMI. Il tutto verrà di seguito chiamato *KMTool*, abbreviazione di *Knowledge Management Tool*.
- Estensione di Firefox – È un plugin del browser Firefox che permette all'utente di collegarsi al KMTool. Verrà di seguito chiamata *FirefoxPlugin*.

Nella pagina accanto, lo schema dettagliato dell'architettura dell'applicazione.



Legenda: nostro software software già esistente protocolli di comunicazione

N.B: Nella nostra concezione, il MetadataServer può accettare le connessioni di più KMTool, mentre il KMTool può connettersi a un solo MetadataServer.

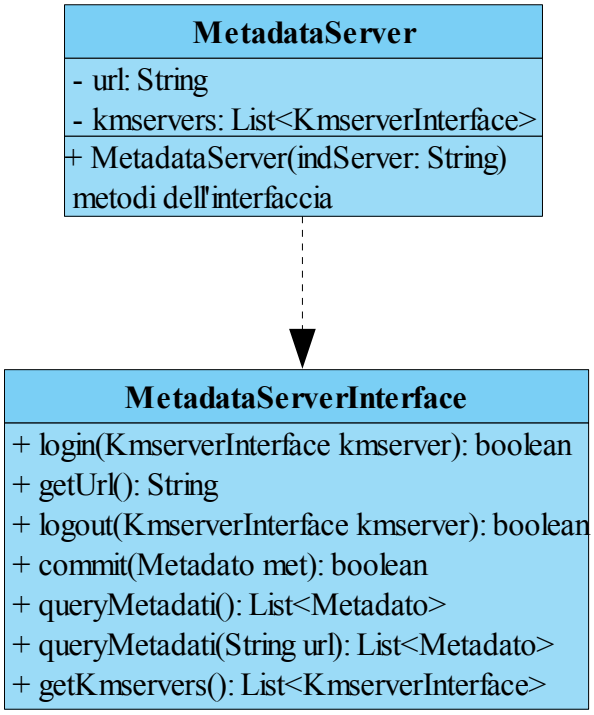
4.2 MetadataServer

La struttura dei package Java e del loro contenuto di questo componente è la seguente:

- [package] connector
  - [classe] Connector
- [package] data
  - [classe] Individuo.java
  - [classe] Metadato.java
  - [classe] OntologyIndividuals.java
- [package] data_db
  - [file di testo] database.createtable
  - [file di testo] database.properties
  - [file di testo] metadati_filler.txt
- [package] filesystem
  - [file di testo] help.txt
  - [file batch] main.bat
  - [file di testo] mypolicy.txt
- [package] kmserver
  - [interfaccia] KmserverInterface
- [package] main
  - [classe] ServerMain.java
- [package] ontology
  - [file ontologia] metadati.owl
- [package] server
  - [classe] MetadataServer.java
  - [interfaccia] MetadataServerInterface.java
- [package] servlet
  - [interfaccia] KmRmiInterface.java

Le parti importanti di questo componente sono l'interfaccia RMI di collegamento (MetadataServerInterface), la classe che implementa questa interfaccia (MetadataServer), la classe main che mette il server in ascolto (ServerMain) e la classe che gestisce il database (Connector). Descriveremo quindi queste classi in dettaglio, riferendoci ogni tanto ad altre classi o file ad esse necessari.

Classe server e interfaccia RMI



Attributi:

- - url: String

  È il campo in cui viene salvato l'indirizzo IP del MetadataServer.

- - kmservers: List<KmserverInterface>

  È la lista dei server di RMI dei vari KMTool connessi.

Costruttori:

- + MetadataServer(indServer: String)

  Costruisce un MetadataServer con l'ip della macchina sulla quale sta andando.

Metodi:

- + login(KmserverInterface kmserver): boolean,
  + logout(KmserverInterface kmserver): boolean

  Metodi per l'autenticazione del server di RMI di un KMTool

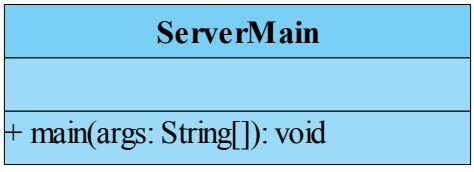
- + commit(met: Metadato): boolean,
  + queryMetadati(): List<Metadato>
  + queryMetadati(url: String): List<Metadato>

  Metodi per la gestione del database. Tutti richiamano metodi omonimi della classe Connector. Il metodo commit invia un metadato al database, mentre queryMetadati recupera tutti i metadati presenti su database (se viene passato il parametro url, queryMetadati recupera i metadati della pagina web con quell'url).

- + getUrl(): String, +getKmservers(): List<KmServerInterface>

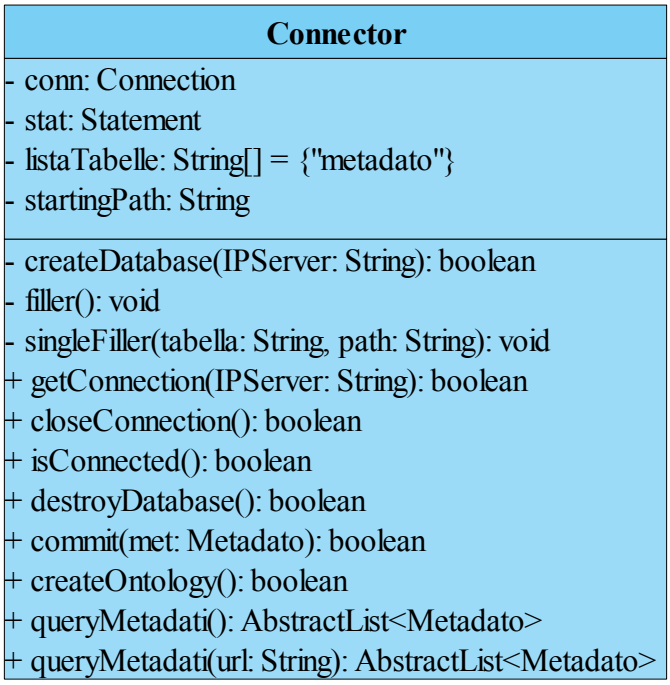
  Sono i getters degli attributi della classe MetadataServer.

Classe main



Il metodo main di questa classe viene lanciato dal main.bat contenuto nel package filesystem. Vengono creati gli stub per la comunicazione via RMI e viene messo il server in ascolto. Digitando help viene visualizzato l'elenco dei comandi disponibili.

Classe Connector e database



La classe Connector permette al MetadataServer di creare il database dei metadati con due metadati di default, che vengono recuperati dal file data_db\metadati_filler.txt. I dati per la connessione al database sono contenuti nel file data_db\database.properties, mentre i dati per la creazione delle tabelle si trovano in data_db\database.createtable.

Poiché il database contiene una sola tabella e non ci sono relazioni, mostriamo qui sotto il solo modello logico tralasciando lo schema ER.

METADATO(ID, URL, stringa, ontologia, individuo)

ID: integer – codice numerico assegnato automaticamente dal database.

URL: varchar(1000) – url della pagina che contiene la parola annotata.

stringa: varchar(45) – la parola (o la frase) annotata.

ontologia: varchar(45) – ontologia del KMTool che contiene il significato della parola.

individuo: varchar(45) – individuo del KMTool da cui recuperare il link automatico.

Per inviare metadati tra MetadataServer e KMTool, è stata creata la classe Metadato nel package data. Siccome Metadato è una classe serializzabile, le sue istanze possono viaggiare in rete. I suoi attributi sono esattamente i campi della tabella metadato del database.

Dopo la creazione del database, è possibile modificarlo utilizzando gli altri metodi disponibili, cioè il commit (per aggiungere un metadato), i due metodi queryMetadati (per recuperare tutti i metadati oppure i metadati di una pagina Web specifica) e il metodo destroyDatabase (per distruggere il database). Riportiamo qui di seguito le istruzioni SQL eseguite da ogni metodo.

+ commit(met: Metadato): boolean

```
INSERT INTO metadato(URL,stringa,ontologia,individuo) " +
"VALUES('" + met.getUrl() + "'," + "'" + met.getStringa() + "'," +
"'" + met.getOntologia() + "'," +"'" + met.getIndividuo() + "')
```

+ queryMetadati(): AbstractList<Metadato>

```
CREATE OR REPLACE VIEW vista_metadati AS SELECT * FROM
`metadato`"
SELECT * FROM vista_metadati
```

+ queryMetadati(url: String): AbstractList<Metadato>

```
CREATE OR REPLACE VIEW vista_metadati AS SELECT * FROM
`metadato` WHERE URL='" + url + "'"
SELECT * FROM vista_metadati
```

+ destroyDatabase(): boolean

```
DROP DATABASE metadati
```

Creazione ontologia a partire dal database

Nel MetadataServer è stata implementata una funzione che permette di creare un'ontologia OWL in ontology\metadati.owl a partire dai metadati contenuti nel database. Il metodo createOntology della classe Connector è proprio deputato a questa funzione, e viene lanciato ogni volta che il database viene modificato da un utente che inserisce un nuovo metadato e ogni volta che il gestore del server digita il comando "ontology" a console. L'ontologia dei metadati così creata è descritta dalla seguente TBOX.

TBOX

haId: METADATO → INTEGER

haUrl: METADATO → STRING

haStringa: METADATO → STRING

haOntologia: METADATO → STRING

haIndividuo: METADATO → STRING

METADATO $\subseteq$ =1haId

METADATO $\subseteq$ =1haUrl

METADATO $\subseteq$ =1haStringa

METADATO $\subseteq$ =1haOntologia

METADATO $\subseteq$ =1haIndividuo

Per rendere possibile questa funzionalità sono state incluse nel MetadataServer le librerie del progetto Jena (jena.sourceforge.net). L'aspetto interessante è che questa ontologia ora può essere integrata facilmente con altre ontologie di metadati.

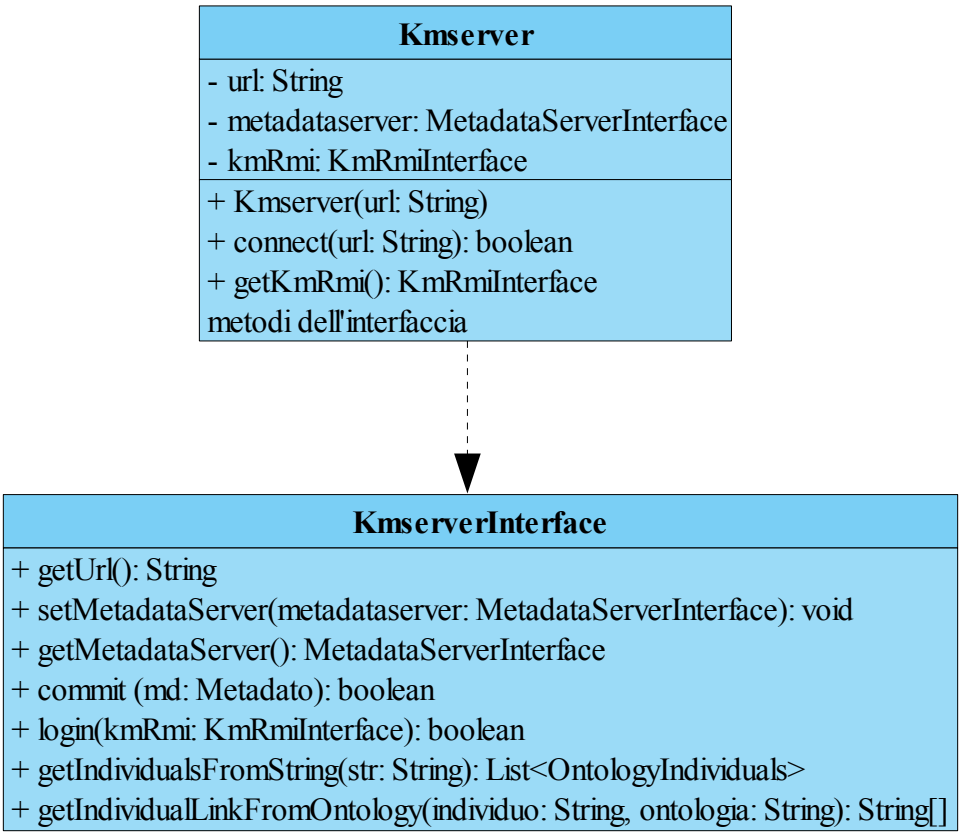
4.3 KMTool

La struttura dei package Java e del loro contenuto all'interno del KMTool è la seguente:

- [package] data
  - [classe] Individuo
  - [classe] Metadato
  - [classe] OntologyIndividuals
- [package] filesystem
  - [file di testo] help.txt
  - [file batch] main.bat
  - [file di testo] mypolicy.txt
- [package] kmserver
  - [classe] Kmserver
  - [interfaccia] KmserverInterface
- [package] main
  - [classe] KmserverMain
- [package] ontologies
  - [file ontologia] Musica_Classica.owl
  - [file ontologia] Politica_Italiana.owl
- [package] ontologyquery
  - [classe] OntologyQuery
- [package] server
  - [interfaccia] MetadataServerInterface
- [package] servlet
  - [classe] AutoLinkServlet
  - [classe] KmRmi
  - [interfaccia] KmRmiInterface
  - [classe] KmServlet

La parte di server RMI che serve per il collegamento con il MetadataServer è rappresentata a livello di classi, interfacce e file dall'interfaccia RMI (KmserverInterface), dalla classe che implementa questa interfaccia (Kmserver), dalla classe main che mette il server in ascolto (KmserverMain), dalla classe che legge le ontologie e ne estrae le informazioni richieste (OntologyQuery) e dalle ontologie (Musica_Classica.owl e Politica_Italiana.owl). La parte di server HTTP a cui si connette la FirefoxPlugin è rappresentata dalle classi AutoLinkServlet (che assolve alla funzione di autolinking) e Kmservlet (che assolve alla funzione di tagging). Il collegamento interno fra queste due parti avviene attraverso la classe KmRmi e la sua interfaccia KmRmiInterface. Una parte importante di questo componente non specificata nella struttura dei package è il sito a cui la FirefoxPlugin si connette durante la funzione di tagging. Esso è contenuto nel file index.jsp che si trova nella cartella WebContent, situata allo stesso livello della cartella che contiene i package (src). Nei prossimi paragrafi descriveremo queste classi e file nel dettaglio ad eccezione del sito index.jsp, che viene mostrato nel Capitolo 5 (Test e Evaluation).

Classe server e interfaccia RMI



Attributi:

- - url: String

  È il campo in cui viene salvato l'indirizzo IP del Kmserver.

- - metadataserver: MetadataServerInterface

  È il MetadataServer a cui il Kmserver è connesso.

- - kmRmi: KmRmiInterface

  È il componente attraverso il quale il Kmserver comunica con i servlet HTTP.

Costruttori:

- + Kmserver(url: String)

  Costruisce un Kmserver con l'ip della macchina sulla quale sta andando.

Metodi:

- + connect(url: String): boolean

  Permette al Kmserver di connettersi al MetadataServer identificato dall'url passato.

- + login(kmRmi: KmRmiInterface): boolean

  Metodo per l'autenticazione del client KmRmi.

- + commit (md: Metadato): boolean

  Invia un metadato al MetadataServer, il quale poi provvede a salvarlo nel database.

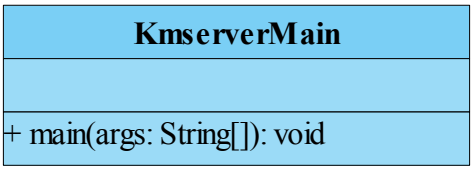
- + getIndividualsFromString(str: String): List<OntologyIndividuals>,

  + getIndividualLinkFromOntology(individuo: String, ontologia: String): String[]

  Metodi che richiamano i metodi omonimi di OntologyQuery per leggere le ontologie ed estrarre le informazioni richieste. Il primo viene utilizzato nella fase di tagging per ricercare all'interno delle ontologie gli individui che hanno il nome uguale alla stringa passata come parametro, il secondo serve nella fase di autolinking per recuperare i link automatici di un individuo contenuto in un'ontologia, che sono i parametri del metodo.

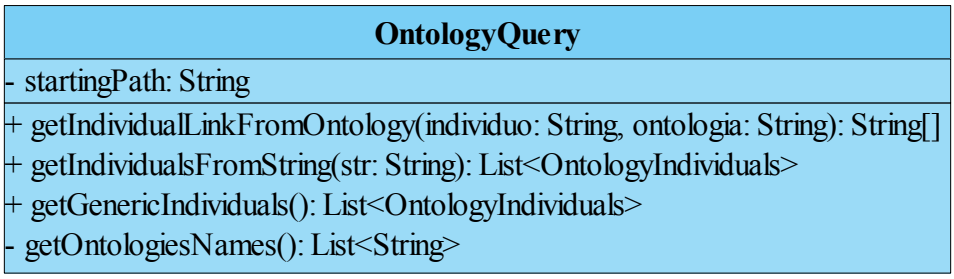
Gli altri metodi sono i getters e i setters degli attributi di KmServer.

Classe main



Il metodo main di questa classe viene lanciato dal main.bat contenuto nel package filesystem, come avviene nel MetadataServer. Vengono creati gli stub per la comunicazione via RMI, viene messo il server in ascolto per le connessioni del KmRmi e viene tentata una connessione automatica al MetadataServer. Se questa connessione non avviene correttamente il Kmserver funziona comunque ma non può inviare né recuperare metadati dal MetadataServer. È quindi necessario che qualcuno da console effettui la connessione manualmente, fornendo l'ip del MetadataServer. Oltre a questa funzionalità, sono disponibili altri comandi contenuti nel file filesystem\help.txt, che viene visualizzato digitando help.

Classe OntologyQuery e ontologie



La classe OntologyQuery mette a disposizione dei metodi al Kmserver per effettuare alcune query sulle ontologie disponibili. Per leggere le ontologie abbiamo dovuto includere nel software le librerie del progetto Jena (jena.sourceforge.net), e per le query in particolare abbiamo utilizzato il modulo ARQ (http://jena.sourceforge.net/ARQ/), sempre contenuto in Jena. Prima di vedere quali query vengono eseguite, vediamo la TBOX delle ontologie utilizzate dal KMTool (Musica_Classica.owl e Politica_Italiana.owl).

Musica_Classica.owl

TBOX

COMPOSITORE ⊆ PERSONA

HaNome: OPERA ∪ COMPOSITORE → STRING

HaLink: OPERA ∪ COMPOSITORE → STRING

HaId: OPERA ∪ COMPOSITORE → STRING

HaOpera: COMPOSITORE → OPERA

HaCompositore: OPERA → COMPOSITORE

COMPOSITORE ⊆ ∃HaNome

COMPOSITORE ⊆ =1HaLink

COMPOSITORE ⊆ =1HaId

COMPOSITORE ⊆ ∃HaOpera

OPERA ⊆ ∃HaNome

OPERA ⊆ =1HaLink

OPERA ⊆ =1HaId

OPERA ⊆ =1HaCompositore

PERSONA ∩ OPERA ⊆ ⊥

Politica_Italiana.owl

TBOX

PRESIDENTE ⊆ PERSONA

HaNome: PRESIDENTE ∪ PARTITO → STRING

HaLink: PRESIDENTE ∪ PARTITO → STRING

HaId: PRESIDENTE ∪ PARTITO → STRING

HaPartito: PRESIDENTE → PARTITO

HaPresidente: PARTITO → PRESIDENTE

PRESIDENTE ⊆ ∃HaNome

PRESIDENTE ⊆ =1HaLink

PRESIDENTE ⊆ =1HaId

PRESIDENTE ⊆ =1HaPartito

PARTITO ⊆ ∃HaNome

PARTITO ⊆ =1HaLink

PARTITO ⊆ =1HaId

PARTITO ⊆ =1HaPresidente

PERSONA ∩ PARTITO ⊆ ⊥

Possiamo ora vedere cosa fanno i vari metodi contenuti in OntologyQuery.

+ getIndividualLinkFromOntology(individuo: String, ontologia: String): String[]

Esegue una query che recupera il link e la classe associati all'individuo dell'ontologia passati come parametro. La query eseguita è la seguente:

```
PREFIX pfx: <" + namespace_ontologia + ">
SELECT ?link ?type
WHERE { pfx:" + individuo + " <" + namespace_ontologia + "HaLink> ?link .
pfx:" + individuo + " <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> ?type . }
```

+ getIndividualsFromString(str: String): List<OntologyIndividuals>

Esegue una query che recupera da tutte le ontologie gli individui che hanno come proprietà HaNome la stringa passata come parametro. La query eseguita è la seguente:

```
SELECT ?x ?type ?id
WHERE { ?x  <" + namespace_ontologia + "HaNome>  \"" + str + "\".
?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type .
?x <" + namespace_ontologia + "HaId> ?id . }
```

Dopo aver eseguito questa query, il metodo lancia getGenericIndividuals.

+ getGenericIndividuals(): List<OntologyIndividuals>

Viene lanciato alla fine di getIndividualsFromString e serve a recuperare da tutte le ontologie gli individui "generici", cioè quelli identificati dal nome "_classe",

come ad esempio l'individuo _COMPOSITORE. Questi individui servono per poter annotare le parole anche nel caso in cui una parola non sia contenuta in un'ontologia. Nell'ontologia della musica classica, per esempio, come compositori abbiamo solo Verdi e Bach. Con l'individuo generico _COMPOSITORE possiamo annotare anche tutti gli altri compositori non contenuti. Il link associato automaticamente a questo generico compositore è il "Mi sento fortunato" di Google, al quale viene attaccato in fondo il concetto COMPOSITORE e la stringa passata. In questo modo, se si vuole taggare Mozart, basterà annotarlo come compositore generico e verrà salvato un metadato sul MetadataServer che avrà come stringa Mozart, come individuo _COMPOSITORE e come ontologia Musica_Classica.owl. La query eseguita per recuperare tutti gli individui generici è la seguente:

```
SELECT ?x ?name ?type ?id
WHERE { ?x  <" + namespace_ontologia + "HaNome> ?name FILTER regex(?name, \"^_\") .
?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type .
?x <" + namespace_ontologia + "HaId> ?id . }
```

Alla fine dell'esecuzione, questo metodo ritorna all'esecuzione di getIndividualsFromString, il quale poi provvede ad unire i risultati della sua query con quella di questo metodo.

Non è ancora stato spiegato il significato della stringa "namespace_ontologia". Questa stringa è il nome univoco assegnato in automatico da Protegé dell'ontologia sulla quale si sta facendo la query. In entrambe le ontologie questa stringa ha come prefisso "http://www.semanticweb.org/ontologies/2007/10/", che significa che l'ontologia è stata creata nel novembre 2007 (10 perché il conteggio parte da 0 = gennaio). In Musica_Classica.owl la stringa termina con "Ontology1195205082538.owl#", mentre in Politica_Italiana.owl termina con

“Ontology1195208539499.owl#”.

Classi servlet

KmServlet
- kmRmi: KmRmiInterface
doGet(request: HttpServletRequest, response: HttpServletResponse): void # doPost(request: HttpServletRequest, response: HttpServletResponse): void - xmlGenerator(individuals: List<OntologyIndividuals>): String

Le classi servlet servono ad accettare le connessioni della FirefoxPlugin e a gestirne le richieste. In fase di tagging, la FirefoxPlugin si collega alla pagina index.jsp del KmServlet e richiede i nomi delle ontologie disponibili e degli individui contenuti in esse, perché per ora in index.jsp ci sono soltanto la parola che l’utente vuole taggare e il sito che contiene quella parola. Una volta che index.jsp contiene anche le ontologie e gli individui, l’utente può selezionare l’ontologia e l’individuo corrispondenti alla parola da taggare. Alla pressione del tasto Submit, la classe KmServlet crea il metadato e lo invia al Kmserver, che poi provvede ad inviarlo al MetadataServer.

Attributi:

- - kmRmi: KmRmiInterface

  È l’oggetto RMI che serve al KmServlet per collegarsi al Kmserver.

Metodi:

- # doGet(request: HttpServletRequest, response: HttpServletResponse): void

  Recupera le ontologie disponibili e i loro individui e li organizza in un file xml tramite il metodo xmlGenerator.
- # doPost(request: HttpServletRequest, response: HttpServletResponse): void

  Legge le informazioni immesse dall’utente nella pagina index.jsp, crea il metadato e lo invia al Kmserver.

- - xmlGenerator(individuals: List<OntologyIndividuals>): String

  Metodo di supporto di doGet. Crea un file xml delle ontologie a disposizione del KMTool e dei loro individui.

AutoLinkServlet
- kmRmi: KmRmiInterface
doGet(request: HttpServletRequest, response: HttpServletResponse): void # doPost(request: HttpServletRequest, response: HttpServletResponse): void - xmlGenerator(metadati: List<Metadato>): String

In fase di autolinking la FirefoxPlugin si collega all'AutoLinkServlet, il quale recupera i metadati della pagina appena aperta in Firefox (se ve ne sono), richiedendoli al Kmserver, che li richiede a sua volta al MetadataServer.

Attributi:

- - kmRmi: KmRmiInterface

  È l'oggetto RMI che serve al KmServlet per collegarsi al Kmserver.

Metodi:

- # doGet(request: HttpServletRequest, response: HttpServletResponse): void

  Metodo non implementato.
- # doPost(request: HttpServletRequest, response: HttpServletResponse): void

  Recupera i metadati della pagina in cui si trova l'utente e li organizza in un file xml tramite il metodo xmlGenerator.
- - xmlGenerator(individuals: List<OntologyIndividuals>): String

  Metodo di supporto di doPost. Crea un file xml dei metadati recuperati.

4.4 FirefoxPlugin

La struttura dei file di questo componente è la seguente:

- [cartella] chrome
  - [cartella] content
    - [cartella] speakinabout
      - [file xul] about.xul
      - [file rdf] contents.rdf
      - [file xul] options.xul
      - [file javascript] speakinaboutOverlay.js
      - [file xul] speakinaboutOverlay.xul
  - [cartella] skin
    - [cartella] classic
      - [cartella] speakinabout
        - [file rdf] contents.rdf
        - [file immagine] speakinabout.png
        - [file immagine] speakinaboutb.png
        - [file immagine] status.png
- [file manifest] chrome.manifest
- [file javascript] install.js
- [file rdf] install.rdf

Il cuore della FirefoxPlugin è costituito dai file contenuti nella cartella chrome\content\speakinabout, e in particolare dal file speakinaboutOverlay.js, che contiene tutte le funzioni javascript utilizzate. Descriveremo questo file nel dettaglio e ci soffermeremo brevemente sugli altri componenti.

File speakinaboutOverlay.js

speakinaboutOverlay.js
function speakinabout(): void function paneInit(): void function pageLoad(event): void function newXMLHttpRequest(): XMLHttpRequest function connect(url): void function genLink(tagSource, stringValue, linkValue): string function tagValue(resText, tag, i): string function strOccurs(resText, str): int function strComp(str1, str2): int

- function speakinabout(): void

  Viene lanciata durante la fase di tagging, quando l'utente clicca col tasto destro sulla parola da taggare e seleziona "Speakin' About". Dopo questa operazione, il browser si collega al sito del KmServlet (index.jsp).

- function paneInit(): void

  Serve a settare in automatico l'ip del KMTool a 127.0.0.1, quando l'utente apre la finestra delle opzioni per la prima volta.

- function pageLoad(event): void

  Viene lanciata all'inizio della fase di autolinking, cioè quando il browser finisce il caricamento di una pagina html. Questa funzione richiama la connect(url), per collegarsi all'AutoLinkServlet.

- function connect(url): void

  Apre un collegamento con l'AutoLinkServlet e richiede i metadati relativi alla pagina che l'utente sta visualizzando, tramite l'utilizzo dell'oggetto XMLHttpRequest, creato dalla funzione newXMLHttpRequest(). Una volta avuti i metadati, questa funzione richiama la funzione genLink per generare un link automatico per ogni metadato.

- function newXMLHttpRequest(): XMLHttpRequest

  Crea un oggetto XMLHttpRequest, che serve a connettersi a un sito e

richiedere dati senza visualizzare alcuna pagina.

- function genLink(tagSource, stringValue, linkValue): string

  Funzione ricorsiva per scorrere la struttura del codice html della pagina. Quando questa funzione trova una porzione di testo, ricerca all'interno di questo testo la stringa stringValue passata come parametro. Se la trova, la sostituisce con il collegamento specificato in linkValue. Per non appesantire l'applicazione, è stato scelto di creare un numero massimo di 10 collegamenti.

- function tagValue(resText, tag, i): string

  Funzione di supporto utilizzata da connect(url) per conoscere il valore del tag dell'i-esimo metadato contenuto nella risposta in formato xml dell'AutoLinkServlet. Viene usato per conoscere i valori dei tags <stringa> e <link>, che vengono poi passati alla genLink.

- function strOccurs(resText, str): int

  Funzione di supporto per contare quante volte nella stringa resText appare la stringa str.

- function strComp(str1, str2): int

  Funzione di supporto per confrontare due stringhe. Restituisce 1 se le due stringhe sono uguali, 0 se sono diverse.

Altri file nella cartella chrome\content\speakinabout

- about.xul

  Specifica i contenuti della piccola finestra che appare quando si clicca in Firefox su Strumenti → Componenti aggiuntivi → tasto destro su Speakin' About → Informazioni su Speakin' About.

- contents.rdf

  File che specifica al browser dove si trova il file speakinaboutOverlay.xul.

- options.xul

  Questo file specifica la grafica della finestra di opzioni dell'applicazione, accessibile cliccando su Strumenti → Componenti aggiuntivi → Speakin' About → Opzioni e grazie alla quale è possibile settare l'indirizzo IP dei servlet.

- speakinaboutOverlay.xul

  Dice al browser dove si trova il codice javascript della FirefoxPlugin (cioè in speakinaboutOverlay.js), qual è la funzione da lanciare ogni volta che viene caricata una pagina (pageLoad) e attraverso quali comandi è possibile attivare l'estensione. Nel nostro caso, può essere lanciata nei seguenti modi:

  - tasto destro sulla parola selezionata → Speakin' About;
  - Strumenti → Speakin' About;
  - immagine  sulla status bar;

File nella cartella chrome\skin\classic\speakinabout

- contents.rdf

  File che indica al browser dove trovare le immagini che servono alla grafica dell'applicazione.

- speakinabout.png

  Logo dell'applicazione che compare cliccando su Strumenti → Componenti aggiuntivi.

- speakinaboutb.png

  Logo dell'applicazione che compare cliccando su Strumenti → Componenti aggiuntivi → tasto destro su Speakin' About → Informazioni su Speakin' About.

- status.png

  Logo dell'applicazione che compare sulla status bar di Firefox.

File allo stesso livello della cartella chrome

- chrome.manifest

  Indica a Firefox di aggiungere l'estensione all'esecuzione del browser.

- install.js

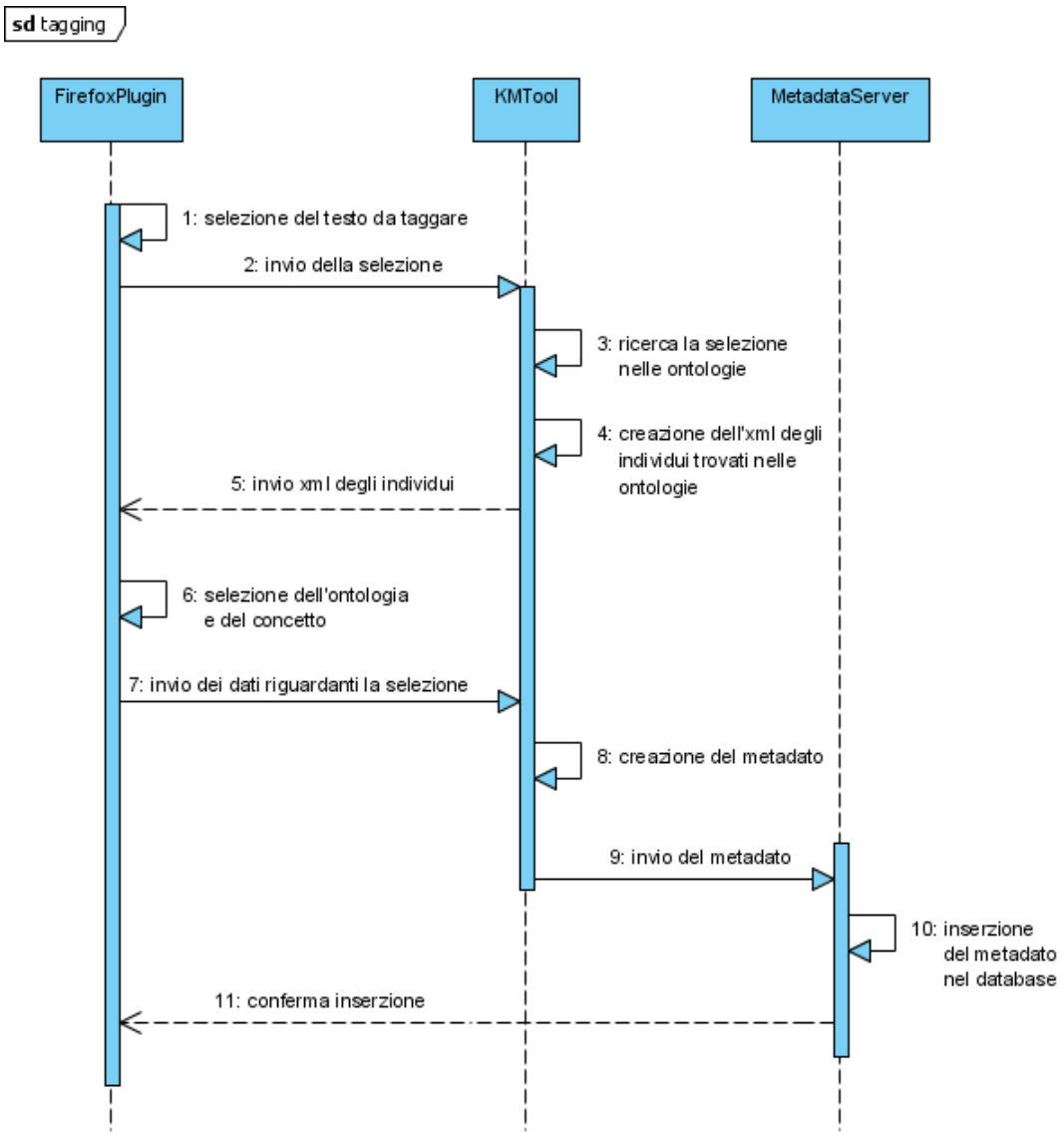
  Installer dell'estensione per versioni di Firefox antecedenti alla 0.9.

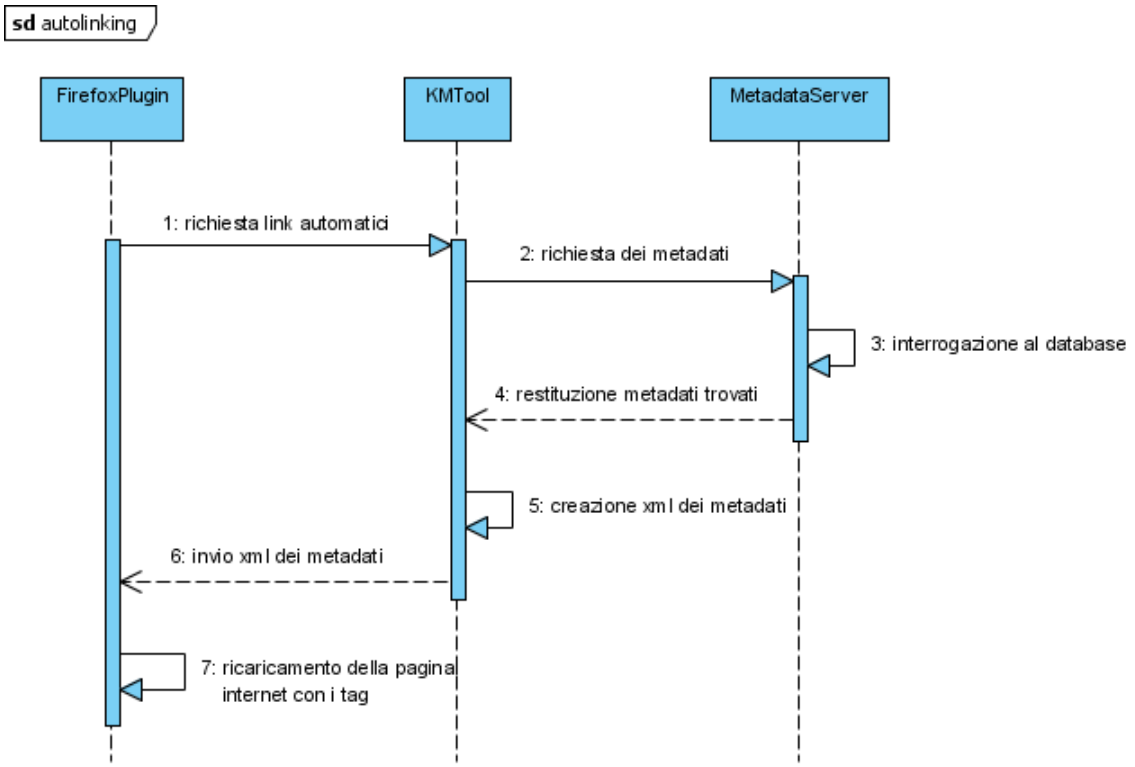
- install.rdf

  Installer dell'estensione per versioni di Firefox superiori alla 0.9.

4.5 Sequence Diagram

Ora che sono stati descritti nei dettagli i tre componenti, possiamo riassumere la loro interazione nelle fasi di tagging e di autolinking attraverso l'utilizzo dei sequence diagram UML.




sd autolinking
FirefoxPlugin
KMTool
MetadataServer
1: richiesta link automatici
2: richiesta dei metadati
3: interrogazione al database
4: restituzione metadati trovati
5: creazione xml dei metadati
6: invio xml dei metadati
7: ricaricamento della pagina internet con i tag

5. TEST E EVALUATION

5.1 Avviare i server

Fino ad ora si è parlato in modo esaustivo del funzionamento dell'applicazione. Ora, tramite alcuni test, vedremo meglio il comportamento del software, in modo da chiarire alcuni aspetti che magari risultavano difficili da apprendere teoricamente.

Per prima cosa, avviamo il MetadataServer. Per fare ciò, eseguiamo semplicemente il file MetadataServer.bat. Dovrebbe aprirsi la shell del server pronta a ricevere comandi – digitiamo "help" per vedere i comandi disponibili – e a ricevere connessioni remote o locali da eventuali KMTool (vedi *Immagine 5.1.1*).

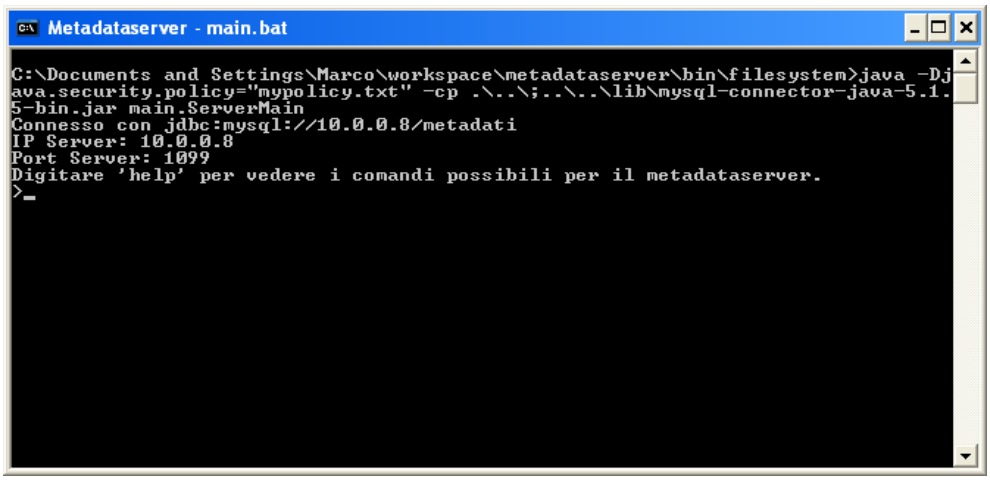


Immagine 5.1.1: *il MetadataServer è avviato.*

L'installazione del KMTool risulterà praticamente identica a quella precedente, solo che in questo caso si eseguirà il file KMTool.bat contenuto nella relativa cartella del server. Non appena avviato, il server tenterà di collegarsi automaticamente in locale al MetadataServer. Se il MetadataServer non è stato avviato sulla stessa macchina, ma su una remota, ci verrà segnalato un errore di connessione. Dovremo quindi fare il login manuale digitando quindi "login" e successivamente inserendo l'indirizzo ip del MetadataServer.

Nei capitoli 2 e 4 si era introdotta la necessità di appoggiarsi alle tecnologie servlet di

Tomcat per permettere all'estensione di Firefox di interagire con il KMTool. Assicuriamoci quindi che Tomcat sia già funzionante.

Passiamo ora ad alcuni esempi di utilizzo del software.

5.2 Esempio di utilizzo

Prima di passare direttamente al test, selezioniamo dal browser "Strumenti" e quindi "Componenti aggiuntivi" e assicuriamoci che nelle opzioni di "Speakin' About" ci sia l'indirizzo IP esatto del server a cui collegarci[1] (vedi *Immagine 5.2.1*).

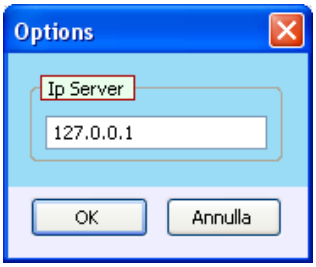


Immagine 5.2.1: *schermata di impostazione dell'indirizzo IP del server.*

Partiamo allora dall'esempio che si era fatto nel capitolo 3, e cioè quello di Giuseppe Verdi. Per prima cosa apriamo il sito http://www.orchestradifiati.it/fiato.php?loc=att_curriculum.htm&a=15 che è una pagina dedicata all'orchestra civica Giuseppe Verdi di Trieste. Come si può ben intuire il nome dell'orchestra fa riferimento al grande compositore di musica classica. Decidiamo allora di *taggare* la parola "Verdi" –in realtà si potrebbe benissimo *taggare* "Giuseppe Verdi" che si avrebbe lo stesso risultato, ma preferiamo *taggare* solo "Verdi" per mettere in evidenza la possibilità di disambiguare il significato della parola. Selezioniamo quindi la parola "Verdi" (eventuali spazi iniziali o finali e caratteri particolari saranno eliminati dalla selezione) e premiamo il tasto destro del mouse (vedi *Immagine 5.2.2*).

1 ATTENZIONE: l'indirizzo IP necessario per il corretto funzionamento dell'applicazione è quello del KMTool e non quello del MetadataServer.

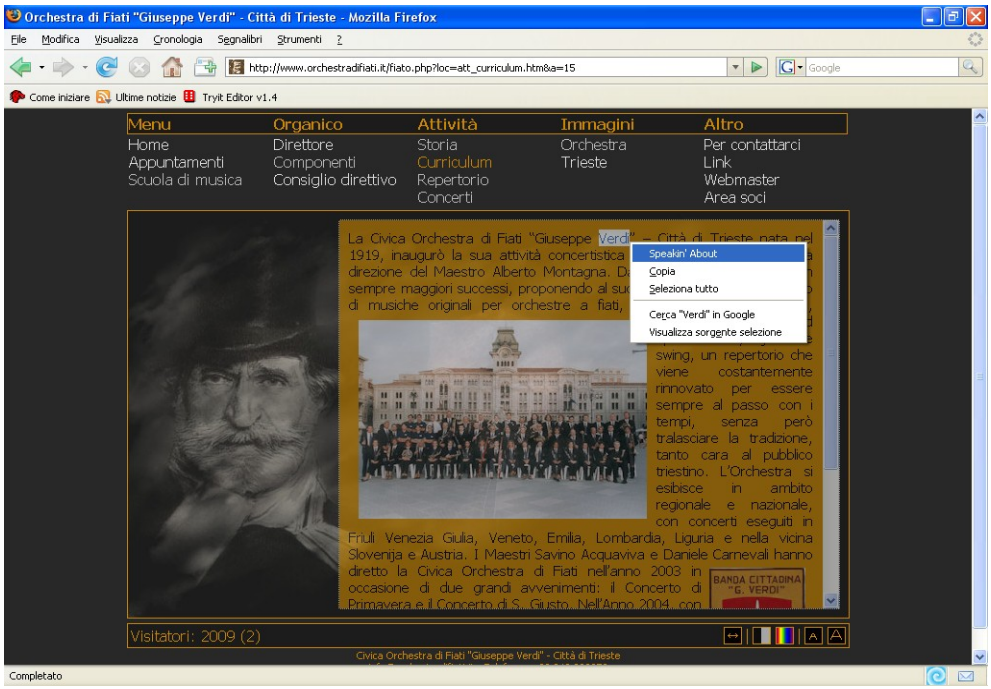


Immagine 5.2.2: *selezione della parola "Verdi" e avvio della funzione di tagging.*

A questo punto apparirà la finestrella del plugin che specificherà la selezione e l'URL della pagina in questione. Subito sotto, si potrà notare una tendina con scritto "Selezionare l'area di interesse..." dalla quale sarà possibile scegliere l'ontologia più adatta. Dato che al momento il KMTool contiene solo due ontologie, appariranno solo queste, e cioè: "Musica_Classica.owl" e "Politica_Italiana.owl" (vedi *Immagine 5.2.3*).

Come ovvio, si selezionerà l'ontologia "Musica_Classica.owl" abilitando in questo modo la seconda tendina, dalla quale sarà possibile scegliere l'individuo specifico. Nel nostro caso comparirà l'individuo "COMPOSITORE - Giuseppe Verdi" e due concetti generici: "COMPOSITORE - ricerca ..." e "OPERA - ricerca ...".

I concetti generici appariranno sempre, ma saranno da usare solo quando non c'è un individuo che corrisponde esattamente al significato che si vuole attribuire alla selezione. Tramite i concetti generici, infatti, si tenterà di ricercare il sito più

appropriato. Per fare ciò, ci si appoggerà al motore di ricerca google[2] (vedi *Immagine 5.2.4*).

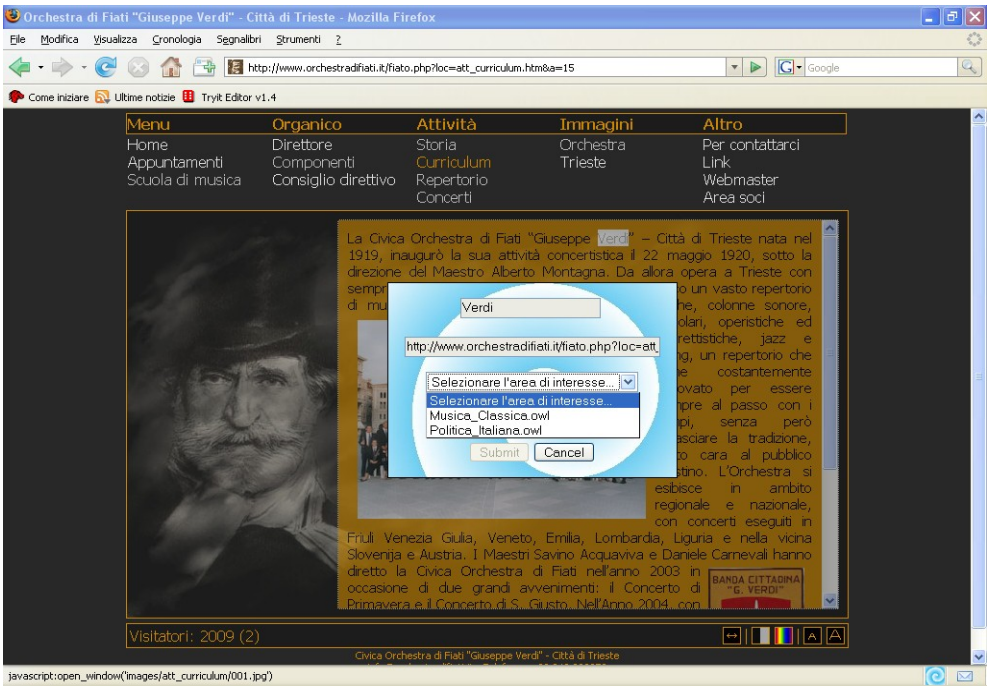


Immagine 5.2.3: *selezione dell'ontologia.*

Nel nostro caso, il concetto da selezionare sarà "COMPOSITORE - Giuseppe Verdi", ma nel caso in cui "Verdi" fosse stato il nome di un'opera (??!?!), si sarebbe potuto scegliere "OPERA - ricerca ..." per tentare di risalire ad un sito che parlasse di questa opera. Dopo aver selezionato il concetto "COMPOSITORE - Giuseppe Verdi", si potrà inviare il nostro metadato al server premendo il pulsante "Submit". Se l'operazione andrà a buon fine, si avrà un messaggio di successo (vedi *Immagine 5.2.5*).

2 Si utilizza la funzione "Mi sento fortunato" di www.google.it.

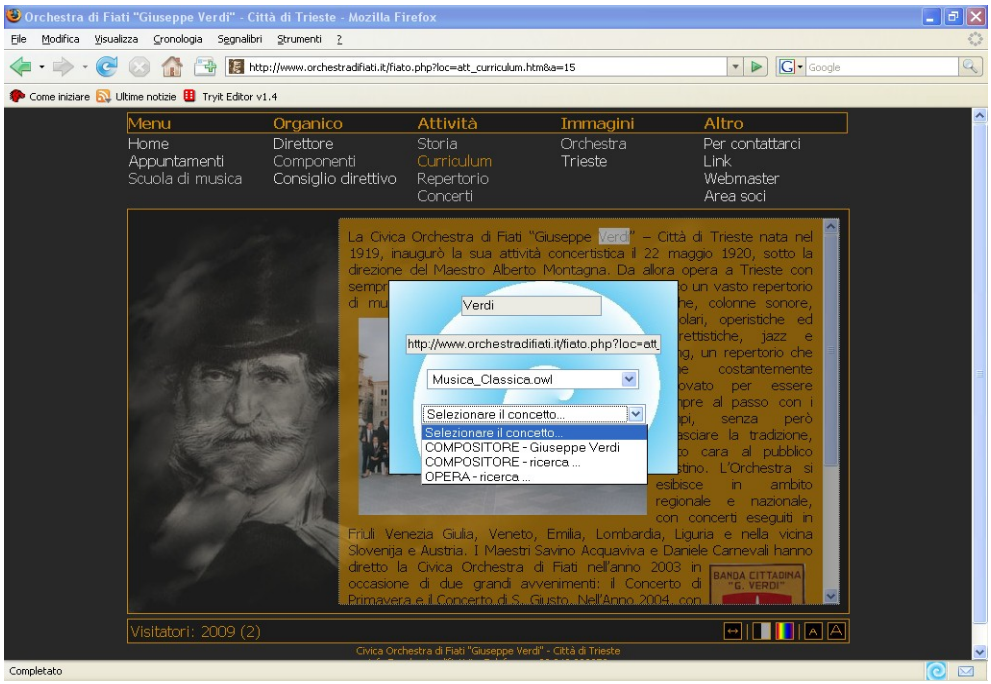


Immagine 5.2.4: *selezione dell'individuo riguardante l'ontologia Musica_Classica.*

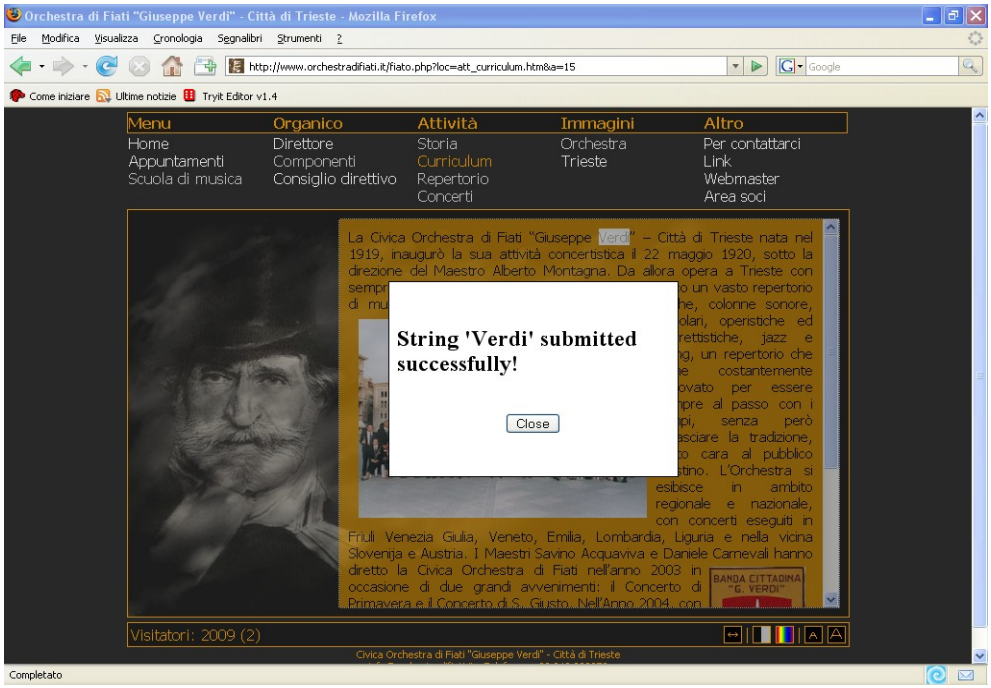


Immagine 5.2.5: *la funzione di tagging è stata eseguita con successo.*

A questo punto è arrivato il momento di verificare la funzione di *autolinking*. Per fare ciò, aggiorniamo la pagina premendo F5. Ora dovrebbe apparire un hyperlink relativo a http://www.giuseppeverdi.it/ sulla parola "Verdi" come nell'*Immagine 5.2.6.*

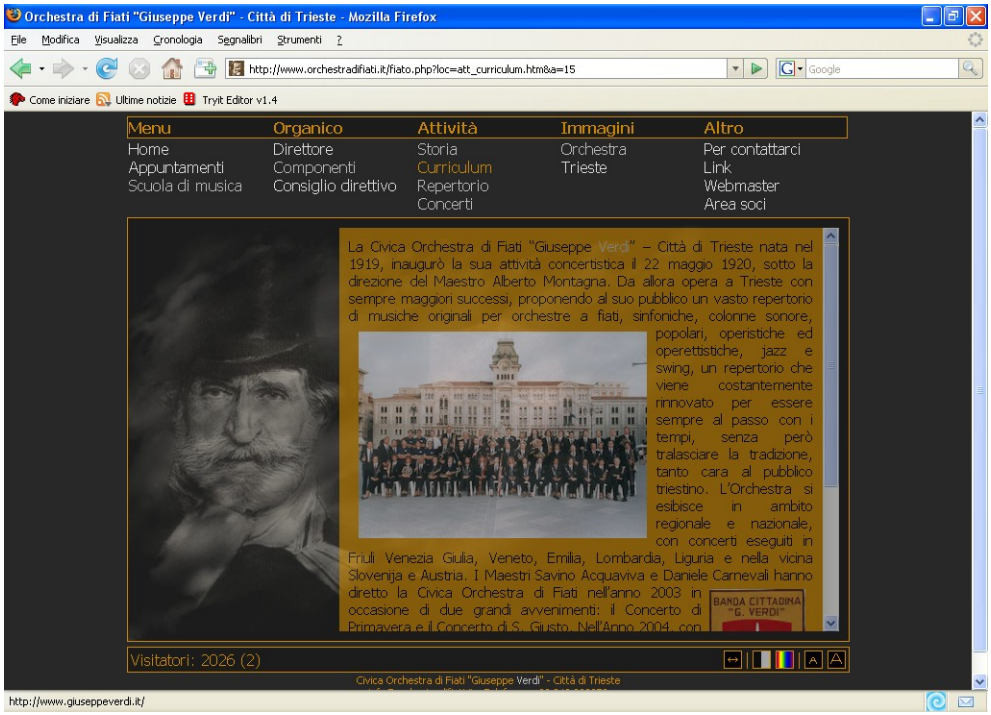


Immagine 5.2.6: *in figura si può notare l'operazione di autolinking.*

5.3 Esempio di utilizzo dei concetti generici

Nel paragrafo 5.3 si era fatto cenno ai concetti generici, e cioè a quei concetti da utilizzare nel caso in cui non si è trovato alcun riscontro della stringa selezionata. Per dirla più semplicemente, in questo esempio tratteremo di quei casi per i quali non è stato possibile trovare il concetto più appropriato in una determinata ontologia.

In questo test rientriamo sempre nell'ambito della musica classica, ma questa volta selezioneremo un compositore che non appare nell'ontologia: Mozart. Come al solito, per prima cosa, apriamo una pagina internet attinente al nostro test. Prendiamo per esempio http://www.mozartitalia.org/ita/storia/storia.html come riferimento.

Selezioniamo ora la parola “Mozart”, premiamo il pulsante destro del mouse e attiviamo “Speakin’ About”. Come si può ben vedere, è ancora possibile scegliere tra le due ontologie “Musica_Classica.owl” e “Politica_Italiana.owl”. A noi interessa la musica classica, dato che Mozart è un compositore, e quindi selezioniamo tale ontologia.

Nella seconda tendina a comparsa si può adesso scegliere l’individuo, ma si nota subito che appaiono solo due concetti generici (che sono “COMPOSITORE - ricerca ...” e “OPERA - ricerca ...”) e nessun individuo. Questo è dovuto al fatto che nell’ontologia non c’è alcuna istanza di “Mozart”. Cosa si fa allora? Selezioniamo il concetto generico “COMPOSITORE - ricerca ...” e, come per il test precedente, premiamo “Submit” per inviare il metadato al server (*Immagine 5.3.1*).

Immagine 5.3.1: *selezione del concetto generico.*

Aggiorniamo ora la pagina con F5; tutte le parole “Mozart” presenti nella pagina saranno legate ad un hyperlink che fa riferimento al motore di ricerca google –in realtà è da precisare che non vengono *taggate* tutte le parole “Mozart”, ma solo le prime dieci

per evitare rallentamenti eccessivi nella fase di *autolinking*. Si utilizzerà quindi google – senza che l'utente svolga la ricerca manualmente– per tentare di risalire al sito più adatto al significato che si vuole dare alla selezione. In questo caso le parole chiave da ricercare saranno "Mozart", che la stringa selezionata, e "compositore", che è la stringa corrispondente al concetto generico. Così facendo, si avrà una alta probabilità di trovare con molta precisione un sito attendibile. Nella *Immagine 5.3.2* si può vedere il riferimento al link nella status bar nella parte inferiore del browser.

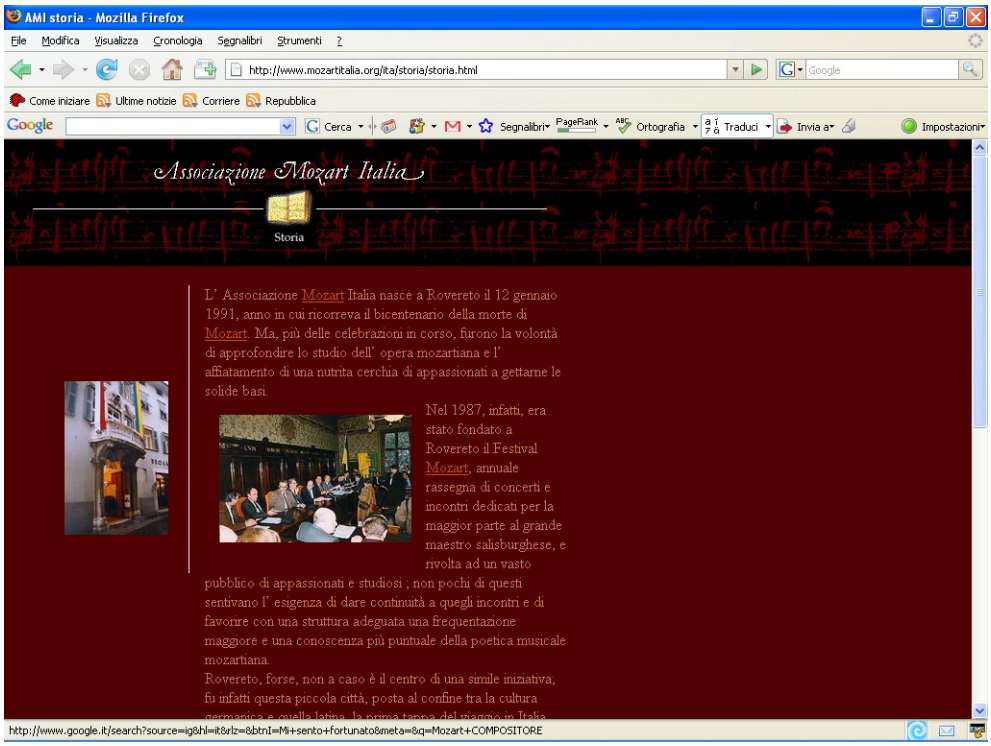


Immagine 5.3.2: *nella status bar è possibile vedere il link a google con le parole da ricercare.*

Proviamo allora a fare un click sulla parola "Mozart" e vediamo dove veniamo reindirizzati. La nuova pagina che si aprirà, comunque, non sarà la classica pagina di ricerca di google con i vari risultati, ma sarà quel sito che google ritiene essere più attinente con la nostra ricerca (vedi *Immagine 5.3.3*).

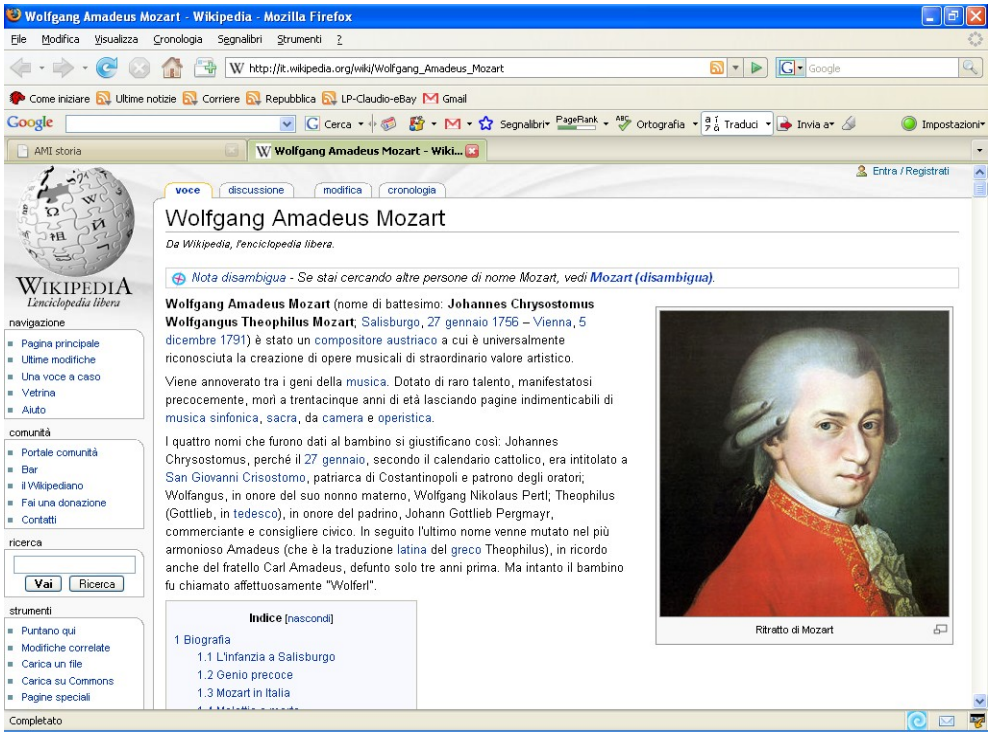


Immagine 5.3.3: *ecco il risultato della ricerca di google.*

Come si può ben vedere, la ricerca di google ci ha portati alla pagina di wikipedia inerente a Wolfgang Amadeus Mozart, ovvero il compositore di musica classica che stavamo cercando.

5.4 Esempio di conversione database-ontologia

Nell'applicazione si è deciso di inserire questa funzione di conversione per scopi didattici e comunque per eventuali evoluzioni del progetto, dato che potrebbe essere utile, in futuro, poter lavorare su un'ontologia anziché su un database. Questa funzione, praticamente, recupera i dati salvati nel database del MetadataServer e crea un'ontologia a partire da questi.

Dato che all'utente che utilizza il plugin non interessa poter usufruire di questo servizio, si è data la possibilità di utilizzo di tale funzione solo all'operatore del server. Questo test, infatti, riguarda solo il lato server del progetto, e non più il lato utente.

Vediamo ora come utilizzare questo servizio.

Dalla console dei comandi del MetadataServer digitiamo la parola “show” per vedere i dati archiviati nel database (*Immagine 5.4.1*).

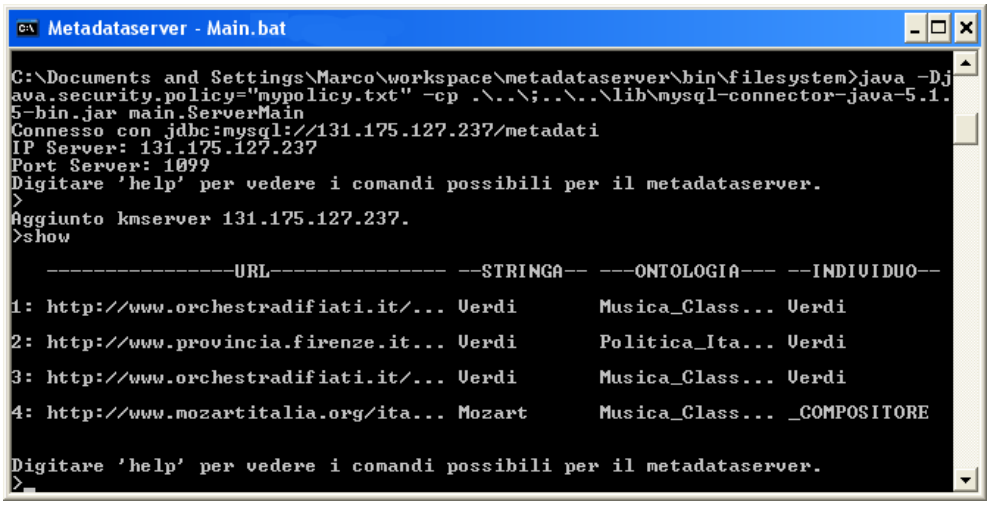


Immagine 5.4.1: *il comando “show” del MetadataServer.*

Nel nostro caso vedremo apparire sul monitor quattro metadati; i primi due che sono quelli caricati di default ogni qualvolta che si crea il database, e gli ultimi due che sono quelli che sono stati salvati in seguito ai due test precedentemente effettuati. Notiamo che il quarto metadato è associato ad un concetto generico, come è giusto che sia.

Ora, per creare l’ontologia, digitiamo il comando “ontology” sulla console. Per poter poi visualizzare l’ontologia, spostiamoci nel percorso ..\metadataserver\bin\ontology dove troveremo il file metadati.owl. Questo file conterrà il risultato dell’operazione di conversione. Aprendo il file con un editor di testo o con un programma per ontologie – ad esempio Protégé– possiamo verificare che l’ontologia risultante è pertinente con i dati che risiedono nel database (vedi *Immagine 5.4.2*).

```
<?xml version="1.0" encoding="windows-1252"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:j.0="http://properties/#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:j.1="http://classes/#" >
  <rdf:Description rdf:about="http://individuals/#metadato3">
    <rdf:type rdf:resource="http://classes/#METADATO"/>
    <j.0:haIndividuo>_COMPOSITORE</j.0:haIndividuo>
    <j.0:haOntologia>Musica_Classica.owl</j.0:haOntologia>
    <j.0:haStringa>Mozart</j.0:haStringa>
    <j.
0:haUrl>http://www.mozartitalia.org/ita/storia/storia.html</j.
0:haUrl>
    <j.0:haId>4</j.0:haId>
  </rdf:Description>
...
...
```

Immagine 5.4.2: *parte dell'ontologia creata con il comando "ontology".*

6 . CONCLUSIONI E SVILUPPI FUTURI

Con la nostra applicazione abbiamo voluto creare un tool per le annotazioni semantiche utilizzabile sul Web. In particolare, si è puntato sulla realizzazione di una interfaccia grafica molto intuitiva e semplice, in modo da consentire un facile approccio all'utilizzo del plugin. Proprio per questo motivo si è scelto di implementare una estensione per Firefox, dato che ci consentiva di creare una applicazione utilizzabile direttamente dal browser, senza il bisogno di dover lavorare con un software a sé stante.

Vediamo ora i principali pregi e limiti del software con annesse possibili espansioni future.

6.1 Pregi e limiti del software

Ora che abbiamo visto le potenzialità del software possiamo fare qualche considerazione a riguardo. I pregi principali del tool sono la facilità di utilizzo di tale software e la possibilità di manutenzione e aggiornamento del software.

Il limite più evidente, però, è data dal fatto che la semantizzazione del web –nel caso di utilizzo del plugin– è dettata dall'utente il quale potrebbe attribuire significati errati alle parole. Ad esempio, se ci troviamo su un sito di automobili e ci imbattiamo nella parola "Panda", possiamo capire immediatamente che fa riferimento alla macchina della FIAT. Se ora utilizziamo il plugin su tale parola, si potrà notare che tra i suggerimenti ci sarà anche l'individuo "ANIMALE – Panda". Chiaramente non è l'individuo adatto al contesto della pagina, ma ci sarà comunque possibile attribuire tale significato anche se ovviamente è errato.

6.2 Possibili sviluppi futuri

Al momento, il software è stato sviluppato nella sua versione base. Nella versione finale, è possibile immaginare l'aggiunta di altre funzionalità come, ad esempio, la creazione di un account, per ogni utente che utilizza l'applicazione, creando un servizio multiutente che permetta la gestione di *tagging* e di *autolinking* in modo differenziato

da utente a utente. Ciò consentirebbe di avere pagine web con link automatici personalizzati.

Un altro possibile sviluppo, potrebbe essere l'aggiunta di template che ci consentano di ricercare la parola selezionata in vari motori di ricerca. Più precisamente, al posto di avere un link ad un unico motore di ricerca nel caso di utilizzo di concetti generici, si potrebbe avere un link ad una pagina sulla quale vengono visualizzati i vari risultati ottenuti da diversi motori di ricerca che possono variare a seconda del concetto preso in considerazione. Ad esempio, se creiamo il metadato contenete la stringa "Il Nome della Rosa", riferito al concetto "Libro", si ricercherà tale stringa usando il template specifico per motori di ricerca di libri –i quali possono essere *http://books.google.it/, http://www.internetbookshop.it/* ecc..–.

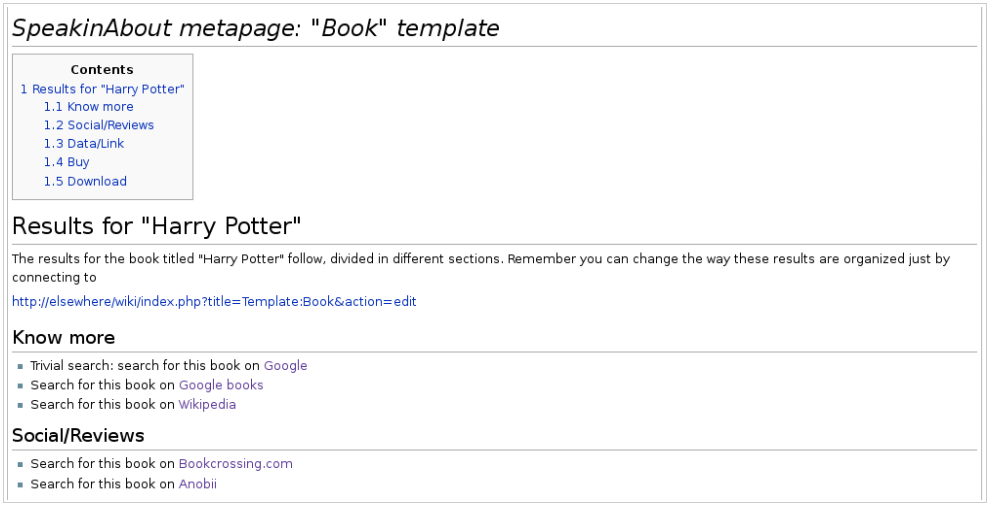


Immagine 6.3.1: *screenshot del template relativo alla ricerca del libro "Harry Potter".*

A tal scopo sono stati sviluppati alcuni template[3] che ci consentono di effettuare le ricerche sopraindicate, anche se al momento non sono ancora parte integrante del nostro plugin. Nella *Immagine 6.3.1* è possibile vedere un esempio di utilizzo di tali template: in questo caso viene ricercata la stringa "Harry Potter" associata alla stringa "Book", poiché si sta utilizzando il template relativo ai libri.

3 Ad opera di Eynard D.

BIBLIOGRAFIA

1. Colombetti M. – Appunti delle lezioni di Ingegneria della Conoscenza.
2. Jos Kahan and Marja-Ritta Koivunen. Annotea: an open rdf infrastructure for shared web annotations. In WWW ’01: Proceedings of the 10th international conference on World Wide Web, pages 623–632, New York, NY, USA, 2001. ACM Press.
3. John Domingue, Martin Dzbor, and Enrico Motta. Collaborative semantic web browsing with magpie. In Christoph Bussler, John Davies, Dieter Fensel, and Rudi Studer, editors, ESWS, volume 3053 of Lecture Notes in Computer Science, pages 388–401. Springer, 2004.
4. Atanas Kiryakov, Borislav Popov, Ivan Terziev, Dimitar Manov, and Damyan Ognyanoff. Semantic annotation, indexing, and retrieval. Journal of Web Semantics: Science, Services and Agents on the World Wide We, 2(1):49–79, 2004.
5. Dill S., Eiron N., Gibson D., Daniel D., Guha R., Jhingran A., Kanungo T., Rajagopalan S., Tomkins A., Tomlin J.A., Zien J.Y. (2003) SemTag and Seeker: Bootstrapping the Semantic Web via Automated Semantic Annotation, In Proceedings 12th International World Wide Web Conference (WWW 2003), Budapest, Hungary, 20-24 May, 2003.

APPENDICE A

A1. Output di help del MetadataServer

```
************************* help *************************

    create: crea il database se non esiste gia'.
    drop: distrugge il database.
    ontology: crea un'ontologia a partire dal database.
    show: visualizza il contenuto del database.
    stat: visualizza lo stato della connessione.
    exit: chiude la connessione ed il server.

*********************************************************
```

A2. Metadati di default

I metadati di default inseriti all'atto della prima creazione del database sono i seguenti:

Metadato1

ID = 1

URL = http://www.orchestradifiati.it/fiato.php?loc=att_storia.htm&a=15

stringa = Verdi

ontologia = Musica_Classica.owl

individuo = Verdi

Metadato2

ID = 2

URL = http://www.provincia.firenze.it/politica/consiglio.htm

stringa = Verdi

ontologia = Politica_Italiana.owl

individuo = Verdi

A3. ABOX metadati.owl

Se nell'ontologia sono i presenti i metadati di default, l'ABOX sarà la seguente.

ABOX

METADATO(metadato0)
METADATO(metadato1)
haId(metadato0, 1)
haUrl(metadato0, http://www.orchestradifiati.it/...)
haStringa(metadato0, Verdi)
haOntologia(metadato0, Musica_Classica.owl)
haIndividuo(metadato0, Verdi)
haId(metadato1, 2)
haUrl(metadato1, http://www.provincia.firenze.it/politica/consiglio.htm)
haStringa(metadato1, Verdi)
haOntologia(metadato1, Politica_Italiana.owl)
haIndividuo(metadato1, Verdi)
≠(metadato0, metadato1)

A4. Output di help del KMTool

```
************************* help *************************

    login: effettua il login sul server di metadati.
    logout: effettua il logout dal server di metadati.
    servlet: mostra le informazioni sul servlet.
    exit: chiude il server.

********************************************************
```

A5. ABOX Musica_Classica.owl

ABOX

COMPOSITORE(Verdi)

COMPOSITORE(Bach)

COMPOSITORE(_COMPOSITORE)

OPERA(Aida)

OPERA(Toccata_e_fuga_in_Re_minore)

OPERA(_OPERA)

HaNome(Verdi, Verdi)

HaNome(Verdi, Giuseppe Verdi)

HaNome(Verdi, Giuseppe Fortunino Francesco Verdi)

HaId(Verdi, Giuseppe Verdi)

HaLink(Verdi, www.giuseppeverdi.it)

HaOpera(Verdi, Aida)

HaNome(Bach, Bach)

HaNome(Bach, Sebastian Bach)

HaNome(Bach, Johann Sebastian Bach)

HaId(Bach, Johann Sebastian Bach)

HaLink(Bach, http://www.jsbach.org/)

HaOpera(Bach, Toccata_e_fuga_in_Re_minore)

HaNome(Aida, Aida)

HaId(Aida, Aida)

HaLink(Aida,http://www.giuseppeverdi.it/page.asp?IDCategoria=162&IDSe...)

HaCompositore(Aida, Verdi)

HaNome(Toccata_e_fuga_in_Re_minore, Toccata e fuga in Re minore)

HaNome(Toccata_e_fuga_in_Re_minore, Toccata e fuga in D min)

HaNome(Toccata_e_fuga_in_Re_minore, Toccata e fuga)

HaNome(Toccata_e_fuga_in_Re_minore, Toccata e Fuga in Re minore)

HaNome(Toccata_e_fuga_in_Re_minore, Toccata e fuga in re minore)

HaNome(Toccata_e_fuga_in_Re_minore, Toccata e Fuga in re minore)

HaNome(Toccata_e_fuga_in_Re_minore, Toccata e Fuga)

HaNome(Toccata_e_fuga_in_Re_minore, BWV 565)

HaId(Toccata_e_fuga_in_Re_minore, Toccata e fuga in Re minore)

HaLink(Toccata_e_fuga_in_Re_minore, http://www.jsbach.org/bwv565.html)

HaCompositore(Toccata_e_fuga_in_Re_minore, Bach)

HaNome(_COMPOSITORE, _COMPOSITORE)

HaId(_COMPOSITORE, ricerca...)

HaLink(_COMPOSITORE,http://www.google.it/search?source=ig&hl=it&r...)

HaOpera(_COMPOSITORE, _OPERA)

HaNome(_OPERA, _OPERA)

HaId(_OPERA, _OPERA)

HaLink(_OPERA,http://www.google.it/search?source=ig&hl=it&rlz=&bt...)

HaCompositore(_OPERA, _COMPOSITORE)

≠(Verdi, Bach, _COMPOSITORE)

≠(Aida, Toccata_e_fuga_in_Re_minore, _OPERA)

A6. ABOX Politica_Italiana.owl

ABOX

PRESIDENTE(Fassino)

PRESIDENTE(Pecoraro_Scanio)

PRESIDENTE(Berlusconi)

PRESIDENTE(_PRESIDENTE)

PARTITO(DS)

PARTITO(Verdi)

PARTITO(Forza_Italia)

PARTITO(_PARTITO)

HaNome(Fassino, Fassino)

HaNome(Fassino, Piero Fassino)

HaId(Fassino, Piero Fassino)

HaLink(Fassino,http://wwww.dsonline.it/partito/organismi/dettaglio.asp...)

HaPartito(Fassino, DS)

HaNome(Pecoraro_Scanio, Pecoraro Scanio)

HaNome(Pecoraro_Scanio, Alfonso Pecoraro Scanio)

HaId(Pecoraro_Scanio, Alfonso Pecoraro Scanio)

HaLink(Pecoraro_Scanio, http://www.pecoraroscanio.it)

HaPartito(Pecoraro_Scanio, Verdi)

HaNome(Berlusconi, Berlusconi)

HaNome(Berlusconi, Silvio Berlusconi)

HaId(Berlusconi, Silvio Berlusconi)

HaLink(Berlusconi, http://www.forzaitalia.it/silvioberlusconi/)

HaPartito(Berlusconi, Forza_Italia)

HaNome(DS, DS)

HaNome(DS, Democratici di Sinistra)

HaId(DS, Democratici di Sinistra)

HaLink(DS, http://www.dsonline.it/)

HaPresidente(DS, Fassino)

HaNome(Verdi, Verdi)

HaNome(Verdi, Partito dei Verdi)

HaId(Verdi, Partito dei Verdi)

HaLink(Verdi, http://www.verdi.it)

HaPresidente(Verdi, Pecoraro_Scanio)

HaNome(Forza_Italia, Forza Italia)

HaId(Forza_Italia, Forza Italia)

HaLink(Forza_Italia, http://www.forza-italia.it/)

HaPresidente(Forza_Italia, Berlusconi)

HaNome(_PRESIDENTE, _PRESIDENTE)

HaId(_PRESIDENTE, ricerca...)

HaLink(_PRESIDENTE,http://www.google.it/search?source=ig&hl=it&rlz=...)

HaPartito(_PRESIDENTE, _PARTITO)

HaNome(_PARTITO, _PARTITO)

HaId(_PARTITO, ricerca...)

HaLink(_PARTITO, http://www.google.it/search?source=ig&hl=it&rlz=...)

HaPresidente(_PARTITO, _PRESIDENTE)

≠(Fassino, Pecoraro_Scanio, Berlusconi, _PRESIDENTE)

≠(DS, Verdi, Forza_Italia, _PARTITO)

A7. xml generato dal metodo xmlGenerator di KmServlet

Se l'utente ha selezionato la parola “Verdi”, e le ontologie disponibili sono Musica_Classica.owl e Politica_Italiana.owl, il file xml creato sarà il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <ontologie>
        <ontologia name="Musica_Classica.owl">
            <individuo name="Verdi">
                <id>Giuseppe Verdi</id>
                <concetto>COMPOSITORE</concetto>
            </individuo>
            <individuo name="_COMPOSITORE">
                <id>ricerca...</id>
                <concetto>COMPOSITORE</concetto>
            </individuo>
            <individuo name="_OPERA">
                <id>ricerca...</id>
                <concetto>OPERA</concetto>
            </individuo>
        </ontologia>
        <ontologia name="Politica_Italiana.owl">
            <individuo name="Verdi">
                <id>Partito dei Verdi</id>
                <concetto>PARTITO</concetto>
```

```
            </individuo>
            <individuo name="_PARTITO">
                <id>ricerca...</id>
                <concetto>PARTITO</concetto>
            </individuo>
            <individuo name="_PRESIDENTE">
                <id>ricerca...</id>
                <concetto>PRESIDENTE</concetto>
            </individuo>
        </ontologia>
    </ontologie>
```

A8. xml generato da xmlGenerator di AutoLinkServlet

Se i metadati sul MetadataServer sono quelli di default e l'utente è sul sito http://www.provincia.firenze.it/politica/consiglio.htm, il file xml è il seguente:

```
<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>
    <metadati>
        <metadato>
            <url>http://www.provincia.firenze.it/politica/consiglio.htm</url>
            <stringa>Verdi</stringa>
            <ontologia>Politica_Italiana.owl</ontologia>
            <individuo>Verdi</individuo>
            <concetto>PARTITO</concetto>
            <link>http://www.verdi.it</link>
        </metadato>
    </metadati>
```

APPENDICE B

B1. Software utilizzato per lo sviluppo dell'applicazione

Eclipse: *http://www.eclipse.org/*

Jena: *http://jena.sourceforge.net/*

Protégé: *http://protege.stanford.edu/*

MySql: *http://www-it.mysql.com/*

B2. Software utilizzato per la stesura dell'elaborato

OpenOffice: *http://www.openoffice.org/*

VisualParadigm: *http://www.visual-paradigm.com/*