

# speech2owl

## Tema d'anno di Linguaggi e Tecnologie Web

Vito Domenico Tagliente

Matricola 567130

Sorgente disponibile su: <https://github.com/vitodtagliente/speech2owl>

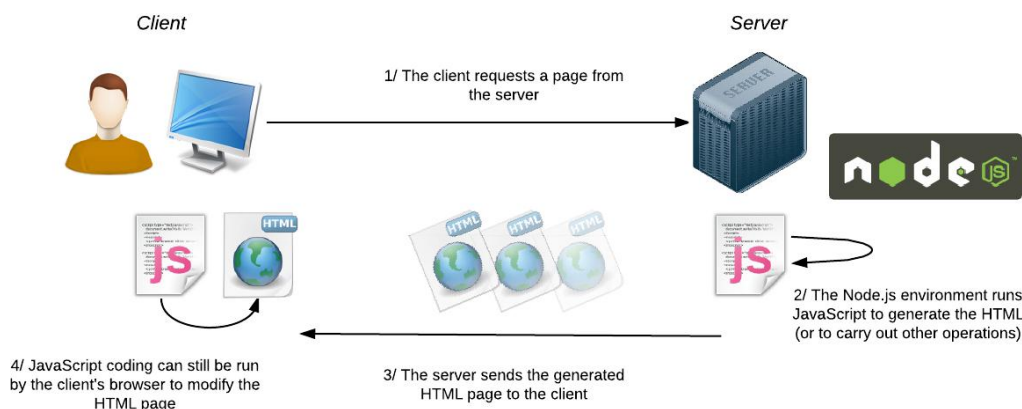
### 1. Introduzione

Scopo del progetto è stato quello di sviluppare una applicazione web-based in grado di creare una annotazione OWL (rispetto ad una ontologia di riferimento) a partire da un messaggio vocale dell'utente.

Per la realizzazione di tale applicazione è stata adottata una architettura client-server basata su tecnologia Node.js.

*Node.js è una piattaforma realizzata su V8, il motore Javascript di Chrome, che permette di realizzare applicazioni web veloci e scalabili. Node usa un modello ad eventi e un sistema di I/O non bloccante che lo rende leggero ed efficiente, perfetto per applicazioni real-time che elaborano dati in modo intensivo e che può essere distribuito su più sistemi.*

*Prima considerazione: fin qua nessuno ha parlato di server, ed in effetti è così! Di solito si liquida Node dicendo che è un "server in javascript", ma la cosa è in parte vera: Node è più in generale "javascript lato server", ovvero è un Motore Javascript, che, tra le altre cose, è capace di tirar su un server HTTP o TCP in pochissime righe di codice! Il server quindi non è qualcosa di scorrelato dall'applicativo web che vogliamo realizzare, ma è una caratteristica dell'applicativo stesso!*



Chiaramente per la realizzazione della parte client dell'applicazione si sono adottate le tecnologie standard per interfacce web, quali HTML5 + CSS + Javascript.



### **Perchè Node.js? Perchè una architettura client-server?**

L'adozione di tale scelta è stata indirizzata dalla disponibilità di numerosi pacchetti, sviluppati dalla community, per ambiente Node. Parliamo di pacchetti che, in ambiente browser, non esisterebbero a causa dei vincoli e dei limiti vigenti. Tutto ciò è stato inoltre guidato da un'ottica di migioria e sviluppo futuro dell'applicazione, in quanto risulterà semplice aggiungere nuove funzionalità, anche complesse, semplicemente installando dei nuovi pacchetti ed aggiornando le API REST dell'applicazione stessa.

## **2. Il Server**

Per quanto riguarda l'implementazione del lato server sono stati utilizzati i seguenti pacchetti:

1. [express](#), un framework per la realizzazione di server HTTP in ambiente Node. Questo permette di definire API REST dei servizi messi a disposizione in tutta semplicità ed in poche righe di codice.
2. [colors](#) è un framework per la colorazione dei log su shell.
3. [pos](#) è il cuore del progetto. Parliamo di un pacchetto di NLP che permette di eseguire lo speech-tagging, una tecnica che consiste nell'esaminare del testo e associare dei tag ad ogni token individuato. Tali tag andranno a specificarne la natura grammaticale (nomi, verbi, etc...).

Nell'implementazione corrente del server, questo si occupa di gestire due sole richieste HTTP di tipo GET. Specificate le routes:

1. [/](#): ritorna l'interfaccia grafica dell'applicazione contenuta nel file [index.html](#).
2. [/nlp](#): accetta del testo come parametro e produce una serializzazione json del risultato del Speech-Tagging eseguito dal modulo pos, prima menzionato.

Per avviare il server occorre eseguire il seguente comando da shell:

```
>> npm start
```

```
menick@menick-X555LPB: ~/Scrivania/speech2owl
menick@menick-X555LPB:~/Scrivania/speech2owl$ npm start

> speech2owl@1.0.0 start /home/menick/Scrivania/speech2owl
> node main.js

speech2owl is listening on port 8000!
```

A questo punto, come già specificato, puntando, con il browser, all'indirizzo `http://localhost:8000`, il server risponderà con l'interfaccia grafica dell'applicazione (ovvero con il contenuto del file `index.html`).

Invece, puntando all'indirizzo `http://localhost:8000/nlp?text=sometext`, si otterrà la serializzazione json del risultato dell'operazione di nlp sul testo specificato.

Richiesta: <http://localhost:8000/nlp?text=hello%20world>

Risultato:

```
[{
  "sentence": "hello world",
  "terms": [
    {"token": "hello", "tag": "UH"},
    {"token": "world", "tag": "NN"}
  ]
}]
```

Ogni tag ha un significato specifico:

NN Noun, sing. or mass	dog
NNP Proper noun, sing.	Edinburgh
NNPS Proper noun, plural	Smiths
NNS Noun, plural	dogs

*La lista completa è riportata nel file `tags.txt`, presente nella root directory dell'applicazione.*

```
menick@menick-X555LPB: ~/Scrivania/speech2owl
menick@menick-X555LPB:~/Scrivania/speech2owl$ npm start

> speech2owl@1.0.0 start /home/menick/Scrivania/speech2owl
> node main.js

speech2owl is listening on port 8000!

NLP::Text = hello world

hello world

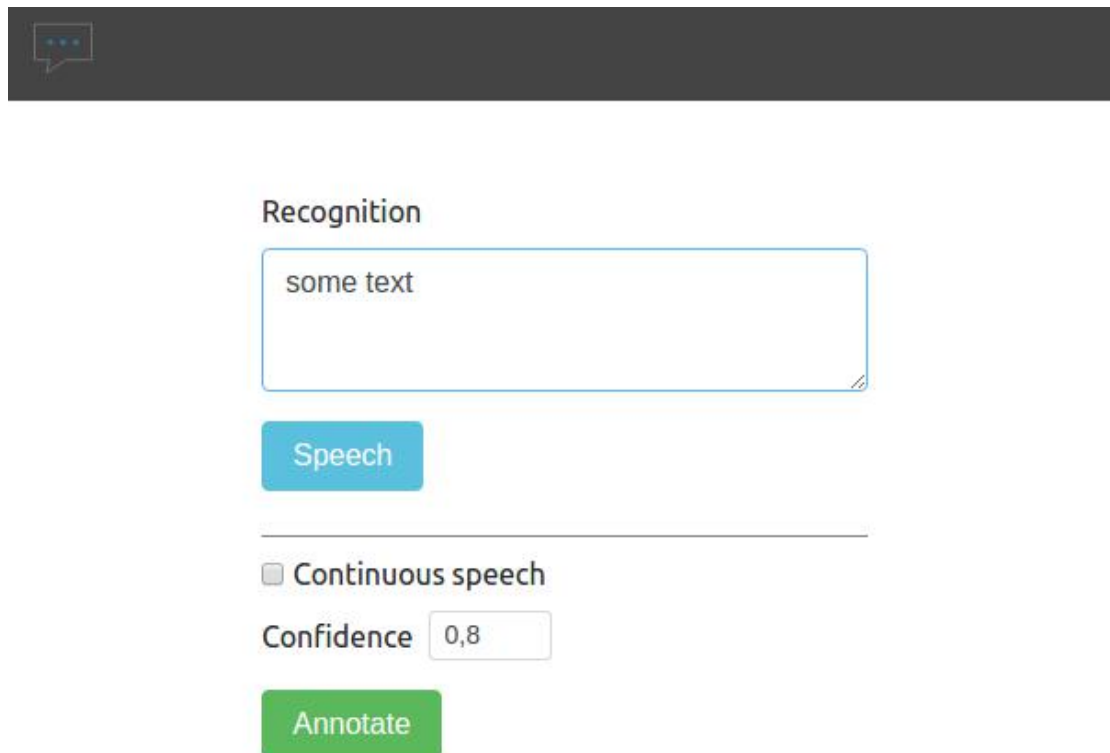
hello /UH
world /NN
```

### 3. Il Front-end

Per quanto riguarda il lato client dell'applicazione, l'interfaccia grafica, come già detto, è stata sviluppata in HTML5+CSS+Javascript.

Per l'impaginazione e la rappresentazione degli elementi grafici è stato utilizzato [Bootstrap](#), la versione 4 alpha. Bootstrap è uno dei framework più utilizzati per lo sviluppo di front-end, in particolare permette uno sviluppo rapido, dovuto alla buona documentazione ed alla numerosa quantità di esempi presenti sul web.

L'interfaccia grafica è stata pensata minimale, presentandosi come segue:



The screenshot displays a web interface with a dark header bar containing a speech bubble icon. Below the header, the main content area is white. It features a section titled "Recognition" in bold. Under this title is a large text input field containing the placeholder text "some text". Below the input field is a blue button labeled "Speech". A horizontal line separates this section from the next. Below the line is a checkbox labeled "Continuous speech", which is currently unchecked. Below the checkbox is a label "Confidence" followed by a small input field containing the value "0,8". At the bottom of the visible area is a green button labeled "Annotate".

Come si nota, questa presenta una textarea per l'inserimento del testo su cui eseguire le operazioni di annotazione. Il testo può essere inserito sia in modalità manuale, input da tastiera, che per riconoscimento vocale. Una volta inserito, è possibile eseguire un'operazione di annotazione dei concetti, rispetto all'ontologia di riferimento, cliccando sull'apposito pulsante, potendo specificare anche il fattore di confidenza.

#### 4. Web Speech API

L'operazione di conversione del parlato in testo è stata implementata utilizzando le API fornite da Google. Tali API sono, ovviamente, disponibili per browser Chrome, anche se, volendo, esiste una implementazione equivalente anche per Mozilla. Per sfruttare tali potenzialità, è stato sviluppato un modulo apposito per un utilizzo semplificato. In particolare, per la realizzazione di questo modulo sono stati applicati due design pattern:

1. Adapter: in quanto mi sono occupato di ridefinire e semplificare l'interfaccia delle API di Google.
2. Singleton: per permettere l'accessibilità semplificata all'istanza dell'oggetto (speech2owl.Speech.singleton).

Quindi, dopo aver incluso lo script necessario:

```
<script src="speech2owl/speech.js"></script>
```

È possibile utilizzare le funzionalità descritte:

```
var speech = new speech2owl.Speech(  
    'speechTextarea', 'speechButton', true  
);  
speech.bindCheckbox( 'speechContinuous' );
```

- ✓ speechTextarea è l'id della textarea in cui verrà salvato il testo riconosciuto.
- ✓ speechButton è, rispettivamente, l'id del pulsante abilitato all'attivazione dell'ascolto.
- ✓ Il terzo argomento è un booleano, opzionale, che serve ad abilitare o disabilitare il permesso di log del modulo.
- ✓ speechContinuous è l'id della checkbox, facoltativa, su cui andare ad abilitare/disabilitare la modalità di ascolto continuo. Nel caso in cui questa non sia attivata, l'intervallo temporale di ascolto delle Web Speech API è limitato.

#### 5. Parsing di Ontologie

Affinchè sia possibile eseguire un match del testo riconosciuto dal modulo di Speech con una ontologia di riferimento, è stato necessario ricercare una libreria per eseguirne il parsing. A tal proposito, la scelta è ricaduta sulla libreria "Simple javascript RDF Parser and query thingy", disponibile all'indirizzo <http://www.jibbering.com/rdf-parser/>.

A questo punto, come fatto per lo Speech Recognition, ho implementato un modulo per la gestione dell'ontologia, utilizzando i pattern Adapter e Singleton.

Nuovamente, occorre includere lo script realizzato:

```
<script src="speech2owl/rdf.js"></script>
```

Per eseguire il parsing di un file RDF/OWL in serializzazione XML occorre:

```
var rdf = new speech2owl.RDF( 'assets/pizza.owl', false );  
rdf.onload = function(){  
    // Loaded, do something  
}
```

A questo punto è possibile utilizzare le API che sono state realizzate per semplificare le operazioni di ricerca e/o matching all'interno dell'ontologia.

Per prima cosa occorre definire i namespace, se necessari:

```
Var RDFS = new  
speech2owl.RDF.Namespace( 'http://www.w3.org/2000/01/rdf-schema#' );
```

Per esempio, per ottenere tutte le possibili triple ?subject ?predicate ?object:  
var results = rdf.any(subject, predicate, object);

Combinando le due funzionalità appena viste, con la seguente istruzione:

```
var results = rdf.any(null, RDFS.get('label'), null);
```

Si ottiene il comportamento equivalente all'esecuzione della query SPARQL:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>  
SELECT ?subject ?object  
WHERE { ?subject rdfs:label ?object }
```

A questo punto è possibile definire anche un match tra le triple ottenute dalla ricerca tramite metodo 'any' ed una generica stringa, potendo definire un fattore di confidenza, compreso tra 0 e 1.

```
var match = rdf.any(subject, predicate, object).match( 'text', confidence );  
Se non specificato, il fattore di confidenza viene valorizzato a 0,8.
```

Il match ottenuto sarà un array di triple, a questo punto è possibile ottenere il match migliore, semplicemente richiamando il metodo 'best'.

```
var best = rdf.any(subject, predicate, object).match( 'text' ).best();
```

## 6. Distanza Jaro-Winkler

Nel precedente capitolo si è parlato di matching tra stringhe. Tale operazione è stata possibile utilizzando l'algoritmo di distanza Jaro-Winkler, che non è altro che un algoritmo che valuta la somiglianza tra due stringhe fornendo un valore numerico compreso tra 0 e 1. Chiaramente, più questo valore è alto, più le due stringhe confrontate sono simili.

Per il progetto corrente è stata utilizzata l'implementazione disponibile all'indirizzo <https://github.com/thsig/jaro-winkler-JS>.

Di seguito viene riportato un esempio di utilizzo:

```
var distance = jaro_winkler.distance( 'text1', 'text2' );
```

## 7. NLP, Natural Language Processing

In questo capitolo esaminiamo il componente più importante del progetto, il modulo di NLP. Tale modulo si occupa di gestire l'output prodotto dal server, in seguito ad una richiesta di Speech-Tagging, e di processare le informazioni ottenute al fine di definire una struttura dati in grado di rappresentare in un modo semplice e compatto la struttura grammaticale del testo esaminato. Per prima cosa occorre includere il modulo di NLP:

```
<script src='speech2owl/nlp.js'></script>
```

Da quanto detto, si intuisce che il testo riconosciuto dal modulo di speech viene inviato al server attraverso una richiesta GET HTTP, richiedendo una elaborazione di Speech Tagging. Tale richiesta viene svolta utilizzando una chiamata Ajax, con una sintassi semplificata grazie all'utilizzo della libreria [jQuery](#) (per il progetto in esame è stata utilizzata la versione 3.1.1):

```
var nlp = null;

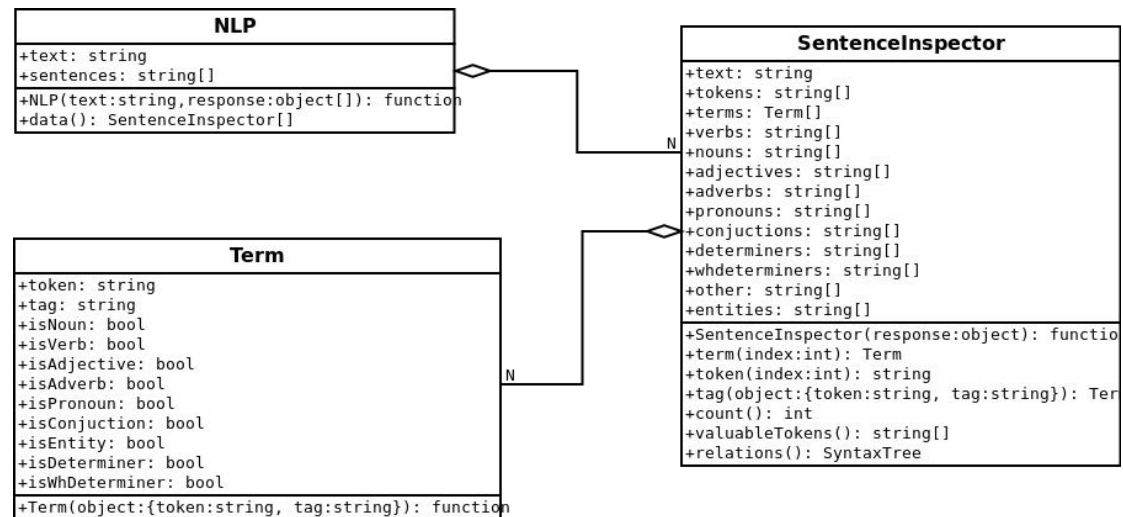
var ajaxRequest = $.ajax({
  'url': 'nlp',
  'data': {
    text: 'hello world'
  },
  'headers': {'Accept': 'application/json'},
  'success': function( response ){
    nlp = new speech2owl.NLP( 'hello world', response );
    // Do something
  },
  'error': function( response ){
    // Do something
  }
});
```

Il server, come già definito nel capitolo 2, risponderà con una serializzazione json dell'output dello Speech-Tagging eseguito:

```
[{
  "sentence":"hello world",
  "terms": [
    {"token":"hello","tag":"UH"},
    {"token":"world","tag":"NN"}
  ]
}]
```

È da notare che l'oggetto restituito è un array di oggetti, in quanto il server si occupa di suddividere il testo, passatogli come argomento, in frasi costituenti e di eseguire lo Speech-Tagging singolarmente su ognuna di queste.

A questo punto esaminiamo da vicino la struttura dei vari componenti del modulo NLP realizzato:



Esaminando il diagramma UML si nota che l'oggetto di tipo NLP si occuperà di istanziare tanti oggetti di tipo SentenceInspector quante le frasi costituenti il testo da elaborare. Questi ultimi, infine, conterranno tanti termini quanti i token individuati nella frase dalla fase di Speech-Tagging, dove in base al tag avremo informazioni dettagliate sulla natura grammaticale di ognuno di questi.

A questo punto è rilevante definire l'importanza del metodo 'valuableTokens' della classe SentenceInspector, ai fini della fase di annotazione delle entità.

Tale metodo diviene disponibile a seguito dell'inclusione del relativo script:

```
<script src='speech2owl/nlp.valuable.js'></script>
```

L'operazione di matching del testo con l'ontologia scelta di riferimento è una operazione che potrebbe essere svolta iterando i token costituenti la frase in esame; così facendo, però, si avrebbe uno spreco di risorse in quanto si andrebbero ad eseguire controlli su token inutili (basti considerare le congiunzioni, la punteggiatura, gli avverbi) e non si avrebbe la possibilità di eseguire un match su parole composte. Pertanto è stato definito il metodo 'valuableTokens' per ovviare a questa problematica, consentendo di concentrare la fase di matching solamente sui token considerati importanti.

Facciamo un esempio, considerando la frase:

John is a boy which likes eating pizza Margherita

Eseguendo un matching basato sui semplici token, andremmo ad iterare e ad eseguire le nostre operazioni di ricerca su ognuno di questi:



```
['John', 'is', 'a', 'boy', 'which', 'likes', 'eating', 'pizza', 'Margherita']
```

Notiamo subito che, anche per una frase così semplice, con un approccio così banale, rischiamo di sprecare tempo e risorse per eseguire il match di token inutili (quali 'a', 'which') e non si otterrebbero mai risultati su token composti. Per esempio, in questo caso, sarebbe utile verificare la presenza nell'ontologia anche delle combinazioni 'likes eating' o 'pizza Margherita'.

Al contrario, richiamando il metodo 'valuableTokens', otteniamo un vettore di token considerati utili, ai fini dell'analisi:

```
[ 'John', 'is', 'boy', ['likes eating', 'likes', 'eating'], ['pizza Margherita', 'pizza', 'Margherita'] ]
```

In questo caso, notiamo che i token vengono combinati in base alla loro natura grammaticale. In seguito, durante la fase di annotazione, sarà l'EntityLinker a capire se è più opportuno considerare valida una loro combinazione o se utilizzare i singoli costituenti.

## 8. Annotazione delle entità

L'operazione di annotazione del testo è stata implementata in un modulo del progetto, denominato EntityLinker.

Tale modulo si occupa di automatizzare l'operazione di estrazione di triple dall'ontologia e di trovare i match migliori.

Come per i moduli precedenti, abbiamo una fase di inclusione:

```
<script src="speech2owl/entitylinker.js"></script>
```

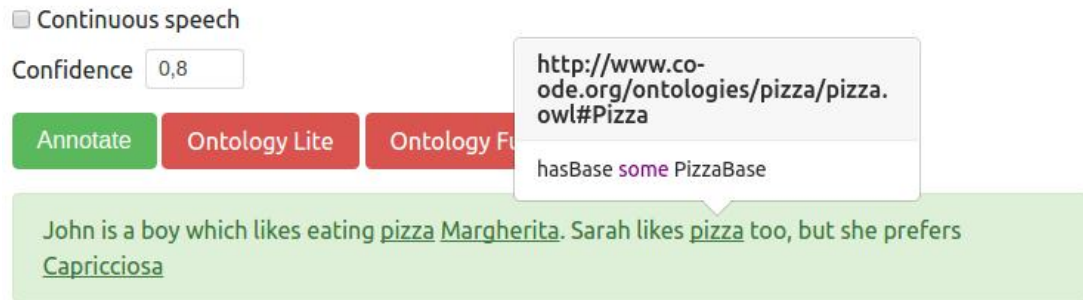
Mentre la fase di annotazione è richiamabile nel seguente modo:

```
var linker = new speech2owl.EntityLinker( nlp, rdf, true );  
var links = linker.link( subject, predicate, object, confidence );
```

- ✓ nlp è un argomento del tipo speech2owl.NLP.
- ✓ rdf è un argomento di tipo speech2owl.RDF.
- ✓ Il terzo argomento è un booleano, opzionale, che serve ad abilitare o disabilitare i log del modulo.

Il metodo 'link' restituisce un array di triple, i match migliori rispetto all'ontologia di riferimento. Anche qui il parametro confidence è facoltativo, se non specificato, questo viene impostato a 0,8.

A questo punto partendo dal testo in chiaro e dalle entità individuate dall'EntityLinker, è possibile eseguire l'annotazione del testo che ci produrrà il seguente risultato (*la parte di codice che svolge questa operazione di rappresentazione grafica delle annotazioni è situata nel file index.html, da riga 175 a 237*):

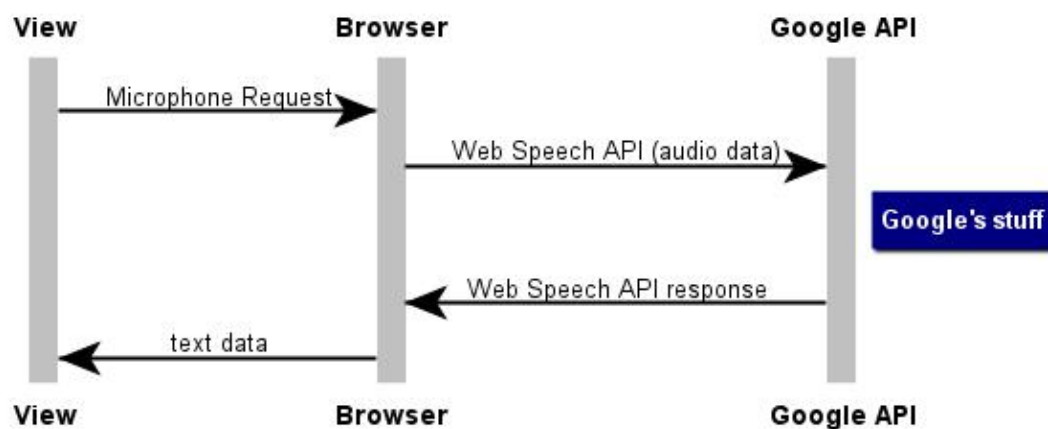


All'interno dei popup rispettivi ad ogni entità individuata, vengono riportate le proprietà ad esse associate, così come definite nell'ontologia di riferimento.

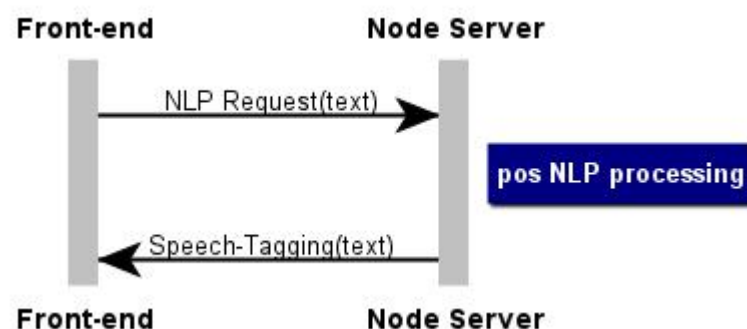
## 9. Riepilogo fase di Annotazione

In questo capitolo verrà riepilogato il funzionamento del processo di annotazione di un messaggio vocale dell'utente, mostrando come i vari componenti, fin'ora descritti, interagiscono per raggiungere l'obiettivo prestabilito.

Il processo parte dalla fase di riconoscimento vocale che, sfruttando le Web Speech API, ci permette di convertire un messaggio vocale in testo. Il funzionamento generale può essere riassunto dal seguente diagramma di sequenza:



Una volta ottenuta la rappresentazione testuale dell'informazione utente, si esegue una richiesta di Speech-Tagging verso il server dell'applicazione:



A questo punto viene inizializzato l'oggetto EntityLinker che in completa autonomia si occupa di eseguire il match migliore tra l'informazione utente e l'ontologia di riferimento.

## 10. Creazione di una Ontologia

Eseguito il matching del messaggio utente con l'ontologia di riferimento, mi sono occupato di definire una rappresentazione owl dei concetti rilevati durante la fase di matching.

A tal proposito mi sono occupato di realizzare una libreria in Javascript per la composizione rapida e agevole di una ontologia OWL.

È da specificare che la libreria non è completa e che, quindi, sono state sviluppate solo le funzionalità richieste per lo scopo prefissato.

Partiamo dalla fase di inclusione degli script (non è stato utilizzato nessun strumento di compilazione e/o di merge dei sorgenti):

```
<script src='owl/owl.js'></script>
<script src='owl/owl.namespace.js'></script>
<script src='owl/owl.ontology.namespace.js'></script>
<script src='owl/owl.ontology.header.js'></script>
<script src='owl/owl.class.js'></script>
<script src='owl/owl.property.js'></script>
<script src='owl/owl.individual.js'></script>
<script src='owl/owl.restriction.js'></script>
<script src='owl/owl.utils.js'></script>
```

In seguito verranno presentate alcune delle funzionalità della libreria realizzata al fine di fornire i concetti fondamentali del suo utilizzo, il resto delle API sarà consultabile nell'appendice di questa documentazione.

La realizzazione di una ontologia parte dalla definizione della stessa:

```
var owl = new OWL.Ontology();
```

Mentre la serializzazione di quest'ultima sarà possibile eseguendo:

```
var ontology = owl.toString();
```

La definizione di una ontologia OWL richiede una fase iniziale di definizione dei namespace che, utilizzando la libreria realizzata, può essere svolta agevolmente e con poche righe di codice:

```
var SPEECH2OWL = new OWL.Namespace(
    'speech2owl',
    'http://www.speech2owl.org/ontologies/speech2owl/speech2owl#'
);

owl.namespace.default().add( SPEECH2OWL.base() );
```

La prima istruzione è banale e si occupa di definire un nuovo namespace per l'ontologia, la seconda invece è un po' più articolata:

- ✓ `owl.namespace.default()`, si occupa di aggiungere i namespace di default per ontologie OWL.
- ✓ Il metodo `'add'` permette di aggiungere anche la definizione del Namespace appena definito, `SPEECH2OWL`.
- ✓ Il metodo `'base'`, richiamato sul namespace appena aggiunto, ci permette di specificare che questo sarà utilizzato come namespace base dell'intera ontologia.

Verifichiamo che, con sole queste due righe di codice, abbiamo prodotto il seguente risultato, richiamabile con `owl.toString()`:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.speech2owl.org/ontologies/speech2owl/speech2owl#"
  xml:base="http://www.speech2owl.org/ontologies/speech2owl/speech2owl"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xml="http://www.w3.org/XML/1998/namespace"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

  <owl:Ontology rdf:about="">
  </owl:Ontology>

</rdf:RDF>
```

A questo punto è possibile definire classi, proprietà, individui, sempre utilizzando delle API compatte ed efficaci. Per esempio, considerando l'ontologia [pizza.owl](#), otteniamo che la seguente definizione:

```
owl.property('#hasBase')
  .functional().inverseFunctional()
  .subPropertyOf('#hasIngredient')
  .range('#PizzaBase').domain('#Pizza')
  .inverseOf('#isBaseOf');
```

È equivalente a:

```
<owl:ObjectProperty rdf:about="#hasBase">
  <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
  <rdf:type rdf:resource="&owl;FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="#hasIngredient"/>
  <rdfs:range rdf:resource="#PizzaBase"/>
  <rdfs:domain rdf:resource="#Pizza"/>
  <owl:inverseOf rdf:resource="#isBaseOf"/>
</owl:ObjectProperty>
```

Per la definizione delle classi, otteniamo che:

```
Ow.class('#CheeseTopping').subClassOf('#PizzaTopping')
    .label('pt', 'CoberturaDeQueijo');
```

È equivalente a:

```
<owl:Class rdf:about="#CheeseTopping">
  <rdfs:label xml:lang="pt">CoberturaDeQueijo</rdfs:label>
  <rdfs:subClassOf rdf:resource="#PizzaTopping"/>
</owl:Class>
```

Stessa cosa se vogliamo definire delle restrizioni:

```
owl.class('#ChickenTopping').label('pt', 'CoberturaDeFrango')
    .subClassOf('#MeatTopping')
    .restriction(
        new OWL.Restriction()
            .onProperty('#hasSpiciness').someValuesFrom('#Mild')
    )
    .disjointWith('#HamTopping')
    .disjointWith('#HotSpicedBeefTopping');
```

È equivalente a scrivere:

```
<owl:Class rdf:about="#ChickenTopping">
  <rdfs:label xml:lang="pt">CoberturaDeFrango</rdfs:label>
  <rdfs:subClassOf rdf:resource="#MeatTopping"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasSpiciness"/>
      <owl:someValuesFrom rdf:resource="#Mild"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#HamTopping"/>
  <owl:disjointWith rdf:resource="#HotSpicedBeefTopping"/>
</owl:Class>
```

Per ulteriori informazioni, visionare la rappresentazione UML posta in appendice.