PROYECTO FIN DE CARRERA

Título:	Desarrollo de un Asistente Personal integrado con un Sistema de Indexación Semántica de Información
Título (inglés):	Design of a personal agent integrated with a Linked Data Indexing System
Autor:	Alberto Mardomingo Mardomingo
Tutor:	Carlos A. Iglesias Fernández
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Mercedes Garijo Ayestarán
Vocal:	Tomás Robles Valladares
Secretario:	Carlos Ángel Iglesias Fernández
Suplente:	

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

PROYECTO FIN DE CARRERA

DESIGN OF A PERSONAL AGENT INTEGRATED WITH A LINKED DATA INDEXING SYSTEM

Alberto Mardomingo Mardomingo

Julio de 2015

42

Resumen

Este proyecto se ha centrado en el diseño y la implementación de un asistente personal integrado con un sistema de indexación semántica de la información, permitiendo la interacción con el usuario mediante el empleo de lenguaje natural.

Para ello, hemos estudiado las tecnologías actuales que nos permiten analizar el lenguaje, recuperar e indexar información semántica y presentársela al usuario.

Así pues, hemos propuesto una arquitectura para nuestro sistema que permitiese llevar a cabo las tareas deseadas, integrando un sistema de pregunta-respuesta, un agente de conversación con capacidad para procesar lenguaje natural, y un sistema de recuperación de la información junto con un un modulo de indexado de dicha información.

Hemos desarrollado varios prototipos para probar nuestra arquitectura, uno de ellos centrándose en un ejemplo sencillo de apoyo a la enseñanza, proporcionando una plataforma para resolver dudas sobre el lenguaje de programación Java, respondiendo las preguntas de los alumnos en castellano, y sugiriendo nuevos temas para que los alumnos profundicen en su estudio.

Otro de los prototipos ha analizado la implementación de un sistema similar para un conjunto heterogéneo de documentos en inglés, poniendo a prueba la capacidad del alumno para diseñar un sistema modular y fácilmente ampliable.

Finalmente, hemos analizado el primer prototipo en un entorno real, recogiendo información sobre la eficiencia del sistema, la tasa de aciertos que presenta ante un corpus de preguntas, a la vez que proponíamos y realizábamos un experimento con usuarios comparando el sistema con interfaces de pregunta-respuesta tradicionales, analizando la mejora en los resultados, la experiencia de uso, y los patrones de comportamiento de los usuarios cuando se enfrentaban a un sistema como el nuestro por primera vez.

Palabras clave: Tecnologías semánticas, Linked data,recuperación de la información, scrapy, scrappy, ChatScript, Solr, WSGI, asistente personal.

Abstract

This project has focused on the design and implementation of a Personal Agent integrated with a Linked Data System, allowing users to interact with it using natural language.

In order to achieve our goal, we have studied current technologies that would allow us to analyse natural language, retrieve and index semantic information, and present it to the user.

We have therefore proposed an architecture for our system that would allow us to perform these tasks, integrating a question answering system, a conversational agent with natural language processing capabilities, as well as an information retrieval system, along with a linked data indexing system.

Based on this architecture, we developed several prototypes that would allow us to test the architecture. The first one of them focus on a simple example of e-learning in Spanish, providing students with a platform to solve their questions about the Java programming language, and offering new topics so they can delve into their study.

A different prototype has analysed the implementation of a similar system for a heterogeneous collection of documents in English, testing the student ability to design a modular system that can be easily extended.

Finally, we have studied our first prototype in a real scenario, gathering information about its efficiency, the success rate while tested with a corpus, as well as proposing an experiment with users. The experiment compared the system with traditional question answering systems, analysing the improvement in the results, the user experience, as well as the patterns in user behaviour when faced with a system like ours for the first time.

Keyword: Semantic technologies, Linked data, Information retrieval, scrapy, scrappy, ChatScript, Solr, WSGI, Personal Assistant

Contents

List of Figures

List of Tables

Listings

Acronyms

BML Behavior Markup Language

IR Information Retrieval

KB Knowledge-Base

NL Natural Language

PoS Part of Speech

QA Question Answering

SPARQL SPARQL Protocol and RDF Query Language

OWL Web Ontology Language

OoB Out of Band

eDisMax Extended Disjunction Max

RDF Resource Description Framework

AIML Artifcial Intelligence Markup Language

LMF Linked Media Framework

WSGI Web Server Gateway Interface

FOAF Friend of a Friend

SIOC Semantically-Interlinked Online Communities

CHAPTER 1

Introduction

In this chapter we will introduce the objectives of this Master Thesis as well as the motivation for them, and describe the structure of this document.

1.1 Context

Personal agents are present in many fields, from educative platforms [1] to virtual city tours [2] or database querying [3]. In this document we present our project, an architecture for conversational systems over linked data, as well as two prototypes, one applied to the educative platforms field, and the other about academic information of a research group. In both cases there has been an increase in the influx of information over the last few years, in a way that made necessary for the end users to have a platform that would allow them to easily access said information.

One alternative for accessing the desired information is using conversational agents, allowing the end user to express their requests and desires in natural language, simplifying the interactions with the system. We will therefore study the technologies available for understanding natural language, as well as information retrieval and indexing technologies, allowing us to build a system that will interact with the users, both answering their questions in natural language, and proposing related topics to look into.

Understanding Natural Language involves using grammars and semantics, statistical methods or templates to identify keywords from the user's input and understand what they are requesting. Maintaining these services whilst reducing the cost and improving their knowledge presents an important challenge for researches, specially when considering than including assistance into the system also involves modelling the new actions the system will have to be able to respond to.

In order to be able to suggest new topics, we will study linked data systems. According to the W3C, the semantic web is "*a Web of Data — of dates and titles and part numbers and chemical properties and any other data one might conceive of*", and Linked Data implies making the data available in a standard format, reachable and manageable by Semantic Web tools, as well as the *relationships among data*, therefore creating a collection of interrelated datasets, that can be referred as Linked Data[1].

Finally, the heterogeneous nature of systems being used by the user, makes it interesting to develop the interface for our system as a web application, allowing access from any platform with access to a web browser.

[1] http://www.w3.org/standards/semanticweb/data

1.2 Goals

The main goal of this Master Thesis is to develop a system that would allow users to interact using Natural Language with a Linked Data System, receiving the information they have asked for, as well as suggestions and information about related topics.

We present a web application that can be used to navigate the information, ask questions and chat with the agent. The system will then be able to find the answer in the Linked Data System, a knowledge base that can be improved using information retrieval techniques.

We will analyse the state of the art systems for Natural Language Processing, in order to chose the most adequate for our system, as well as the systems capable of storing linked data, to be able to select the one that best fit our necessities.

Furthermore, we will study the implementation of our system for two different knowledge fields, the first one with a simple collection of similar documents, and the other with multiple data sources and document structures, but both of them presenting a similar interface for the end user.

Finally, we will evaluate our system, both taking measures regarding its general performance, and designing an experiment with real users to study their responses and experience with our system.

1.3 Structure of the document

In this section we will provide a brief overview of the structure of this Master Thesis. Each chapter is as follows:

Chapter 1 provides an introduction to the project, explaining the basic concepts, as well as the goals for the project.

Chapter 2 list all the technologies used in this project, along with some other systems that are thematically related to the one proposed in this Master Thesis.

Chapter 3 describes the architecture proposed for our system, justifying and explaining each module and its functions.

Chapter 4 shows the functionality of one of the prototypes developed for the project, a system to facilitate the search of information about the Java programming language, explaining the function of each module as well as the interactions between them.

Chapter 5 demonstrates a prototype for a different knowledge field, the information about the Intelligent Systems Group[2], its members, publications and projects.

Chapter 6 discuss the results of the first prototype, showing the performance and analysing the methodology and results of a test with real users.

Chapter 7 analyses the overall result of the Master Thesis, summarizing the goals, and considering possible future developments.

[2]http://www.gsi.dit.upm.es/

CHAPTER 2

Enabling technologies

In this chapter, a brief introduction of the state of the art for conversational agents and Question Answering is presented. Likewise, we will take a short look at some Linked Open Data systems, and the ways to recover data for them.

2.1 Overview

Conversational agents, presented in section 2.2, are systems that allow users to interact with them using natural language, the same way they would interact with another human being. This is achieved by using engines that analyse the user input, process it, and provide the best possible answer given the knowledge of the system.

Question answering systems work in a similar way, but rather than provide a response in natural language, they present the user the resource where the answer to their question is located, usually by translating the question to a specialised query for a given database.

The aforementioned database is usually a Linked Open Data System. This systems allow the publication of Semantic Data, connecting it to the world and therefore making it easily accessible and linkable.

Finally, we will study the way of populating the system, using web scrapping techniques in order to recover the information when it is not presented as Linked Open Data.

2.2 Conversational Agents

In this section we will discuss the evolution of conversational engines, and present a few of the techniques and technologies utilized implementing them. We will provide a small overview to AIML and some engines implemented with it, to finalise with a small description of ChatScript.

One of the starting points when studying conversational agents is A.L.I.C.E. an free natural language artificial intelligence chat robot that utilizes AIML for creating responses based on the user input to the system. ALICE won the 2000, 2001, and 2004 Loebner prizes, becoming a starting point while developing conversational agents. It makes use of the pattern-matching ability of AIML, with 120.000 patterns that can either trigger a response or redirect the input to another pattern. ALICE was inspired by Eliza, on the first examples of natural language processing using simple patterns, written at MIT by Joseph Weizenbaum between 1964 and 1966.

Along with ALICE, a number of other AIML conversational agents have been presented to the Loebner contest, by different authors, usually getting good results, like Mitsuku, by Steve Worswick, who won the 2013 edition of the contest, and was among the 4 finalists in 2014, three of them using AIML. Another example of an AIML bot is Izar, by Brian Rigsby, who achieved second place in the 2014 contest

The winner of the 2010, 2011 and 2014 contests was Bruce Willcox, using different Chatbots, all of them written in ChatScript. Chatscript was presented in 2010, written in C++, and later released as Open Source. Whilst AIML aims to pattern-match words, ChatScript claims to match in a general meaning basis, focusing on detecting equivalence and paying heavy attention to sets of words and canonical representation, and providing a simple way of storing user data, in a machine readable format.

2.2.1 AIML

AIML [4] is a widely used XML dialect for creating conversational language. It was developed between 1995 and 2002 by Richard S. Wallace and the free software community, and has remained relevant to this date, including the draft for a major upgrade, AIML 2.0. This draft was released in the early 2013, and is currently being worked on. The original version of AIML had seven design goals, stated in its primer:

1. Shall be easy to learn.
2. Shall encode the minimal concept set necessary to enable a stimulus-response knowledge system modelled on that of the original A.L.I.C.E.
3. Shall be compatible with XML.
4. It shall be easy to write programs that process AIML documents.
5. AIML objects should be human-legible and reasonably clear.
6. The design of AIML shall be formal and concise.
7. AIML shall not incorporate dependencies upon any other language.

There are more than twenty tags for the AIML language [5], but the most important units are *aiml*, the tag that defines a document as AIML, *category* marking a "unit of knowledge" in the bot's knowledge base, *pattern*, containing a simple pattern that will be compared with the user input, and *template* containing the response to a user input.

```
<category>
   <pattern> WHO ARE YOU </pattern>
   <template>I am a bot </pattern>
<category>
```

Listing 2.1: Example AIML code

The free A.L.I.C.E. AIML [1] includes a knowledge base of 41.000 categories, and can be used as a base for others bots.

2.2.1.1 AIML 2.0

In January 2013 the ALICE A.I. Foundation released a draft specification for a major update for AIML[2], aiming to provide new features while trying to keep AIML as simple as possible. AIML 2.0 combines Pandorabots' extensions to the language, Out of Band tags and a collection of new features. The full list of new features can be found in the Working Draft [6], some of the most relevant are:

- Zero+ wild-cards, allowing to match zero or more words
- Setting matching priority for certain words.
- Loops.
- Out of Band tags.
- Local variables.

2.2.1.2 AIML implementations

AIML has been implemented in multiple languages, including Java, Python, PHP or C++. Some of those implementations are listed in the ALICE downloads page [3], and we will briefly describe some of them bellow:

1. **Program D** is a Java implementation, open source, implementing the AIML specification. It supports multiple bots per instance, and provides multiple ways to interfaces to interact with the service, providing a J2EE release allowing deployment as a web service. However, the development of this project has been stagnated for years, with its last release in 2006.
2. **ChatterBean** is another Java implementation, aiming to be AIML 1.0.1 compliant, using a JavaBeans plug-in architecture, and released under a GPL license. However, the last version was released on 2006 and the development since then seems to have stopped.

[1] http://www.alicebot.org/downloads/

[2] http://alicebot.blogspot.com.es/2013/01/aiml-20-draft-specification-released.html

[3] http://www.alicebot.org/downloads/programs.html

3. **Program O**, written in PHP with MySQL, is an AIML engine with a web interface providing a number of remarkable features, including an administration panel, with configuration, teaching an testing interfaces. It stores the AIML files in a MySQl database in order to improve its performance, and can assign different personalities to each bot. It is under active development by Ellizabeth Perreau and Dave Morton, with it latest version released in May 2014.

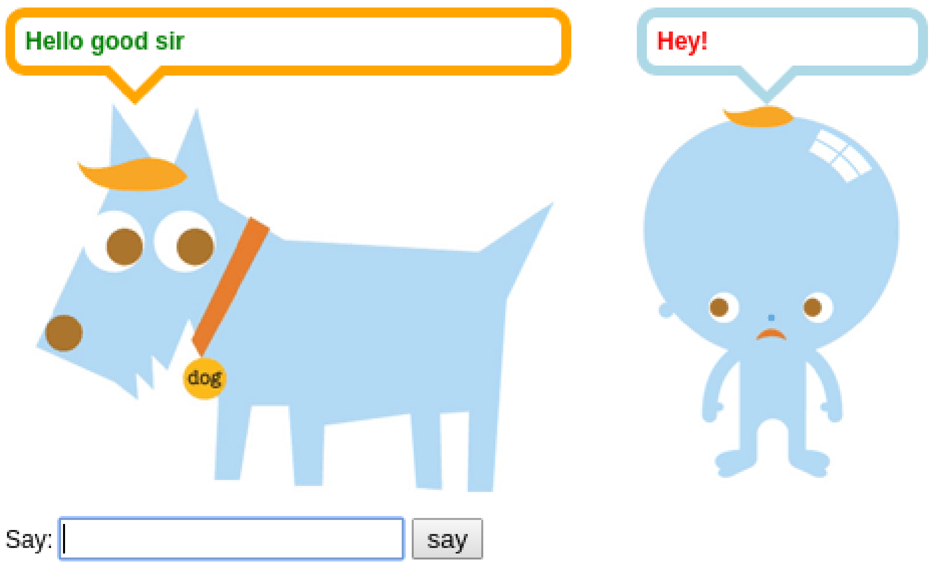


Figure 2.1: Demo interface for Program-O.

2.2.2 ChatScript

ChatScript, written in C++ by Bruce Willcox, is a chatbot engine aiming to overcome the limitations of AIML by providing pattern matching on general meaning rather than particular words. It was originally created for Avatar Reality, a start-up company that wanted to create a virtual world called Blue Mars [7], and it was eventually released as open source as per the requirements for the Loebner prize.

Since then, ChatScript has been updating and introducing improvements, up to version 5.4 [8]. Some of the features in ChatScript include:

- Efficient, easily readable output rules
- Zero-length wild-cards

- Concise pattern matching through the use of concepts
- Built-in data covering multiple subjects and topics

By matching in meaning, ChatScript claims to be able to provide a better user experience than AIML, with a much smaller rule set, improving maintainability and ease to modify the bot whenever necessary.

2.2.2.1 Basic syntax, topics and rules

In Chatscript, rules are grouped in topics, and stored in ".top" files, allowing the designer to map the point of the conversation, therefore making a better response for the user. An example topic file can be found in listing 2.2. In this example, the topic will be triggered when the user writes an input containing the words "name", "here", "what", or any other word specified in the system dictionaries "emogoodbye", "emohello" or "emohowzit". Then, it will attempt to match the user input with the patterns "what is your name" and "what be". If no pattern matches, it will output the gambit "Have you been here before?".

```
topic: ~INTRODUCTIONS (~emogoodbye ~emohello ~emohowzit name here what
    )

#!x issued only once, then erased
t: Have you been here before?

#! what is your name
u: ( what is your name )
    My name is Harry.

#! what are you ?
?: ( what be )
    I am a bot. Are you also a bot?
    a: (~no)
        Oh, a human... How can I help you?
```

Listing 2.2: Example topic file for ChatScript

In this simple example, we find some of ChatScript syntaxes elements. We can see responders, gambits and rejoinders, as well as the use of topics and concepts. These are part of ChatScript rules structure, but there are some more elements not shown here. A slightly more comprehensive list is as follows:

- **Input matching:** The pattern can start with either "s:", "?:" or "u:", indicating they are supposed to respond to statements, questions or both. The parenthesis then indicate the pattern itself, which can have multiple elements:

  General words: A simple word to look for on the sentence. ChatScript handles plurals and verb conjugations.

  Concepts: Starting with ∼ , a concept is a set of words with a common general meaning. There are a number of concepts built-in, and they can be expanded.

  Wild-cards: Allowing to match any word, or using a cardinal, can specify the number of words to match with the wild-card.

  Start and end of sentence: Using > and <, its possible to indicate whether the pattern should match the start or the end of the sentence. << and >> will match any position in the sentence.

  Variables: Starting the pattern element, whether is a wild-card, a concept or a word, using "_", will store the matched word in a variable for future use, be it in the response or somewhere else.

- **Gambit:** Lines starting with "t:" indicate a gambit, that will be triggered when the bot cannot find an appropriate response.

- **Bot Response:** Immediately after the pattern, the following lines until the next ChatScript syntax element indicate how the bot response is formed.

- **Variables:** Not shown in the previous example, any word starting with "$" is a global variable, and can be assigned a value that will be remembered for the user. If the variable starts with "$$", the variable will be local, only lasting until the end of the current answer.

- **Rejoinders** Starting with single letters from "a:" through "q:" to indicate nesting depth, these indicate patterns that will trigger when the user responds to a specific bot output.

- **Concepts:** Sets of words with a specific meaning, when referring to a concept in a pattern, it will match if any word in the set matches the input. ChatScript includes a large set of concepts by default, but the user can define its own sets.

- **Topics:** rules are grouped in topics. Rules inside the topic will only be tested against the input when the input itself contains any of the words specified while defining the topic.

A more detailed approach to ChatScript rules, matching and functions can be found in its documentation[4].

2.2.2.2 Deploying a bot with ChatScript

By default, ChatScript can be run in local or server mode. Local mode allows for direct interaction from the command line, using the standard input and output to communicate with the user. This mode is specially useful for debugging the rules when developing bots, since it gives access to debugging commands built with the system.

For production environments, ChatScript can be run in server mode, where it listens it offers a TCP interface. This interface will receive messages containing the bot name, the user name and the user input, and return the bot response for the given input.

2.3 Question Answering Systems

Question Answering (QA) is a discipline concerned with automatically provide answers to questions presented in natural language, using a number of different approaches in order to process the question into a query the system can understand, and, therefore, answer.

In general, we can differentiate six major general approaches [9]:

- **Controlled natural languages:** The system only takes into account a well-defined subset of a given natural language that can be unambiguously interpreted.
- **Formal grammars processing:** Relaying on linguistics to assign syntactic and semantic representations to lexical units, as well as compositional semantics, these systems compute a representation of the question. Two examples of this approach could be ORAKEL [10] and Pythia [11]
- **Mapping linguistics to semantic structures:** Systems designed under this principle rely on a measure of similarity between elements in the query and the predicates, subjects or objects in the knowledge base. PowerAqua [12] and Aqualog [13] are two examples of this approach.
- **Template-based:** Taking two stages, this approach first constructs a query based on the linguistic analysis of the input question, and the matches the expressions in the question with elements from the dataset. TBSL [14] implements this approach.

[4]`http://sourceforge.net/projects/chatscript/files/`

- **Graph exploration:** This approach maps elements of the question to entities in the knowledge base, and proceeds navigation from these pivot elements to navigate the graph, seeking to connect the entities to yield a connected query. This example is taken by the TREO [15] system
- **Machine learning:** Question Answering has been considered a machine learning problem, with either models for joint query interpretation and response ranking, aiming at learning semantic parsers given a knowledge base and a set of questions and answers, or systems with an algorithm for matching natural language expressions and ontology concepts, as well as an algorithm for storing matches in a parse lexicon.

To the aforementioned systems, we have to add another one: IBM Watson's DeepQA. Designed to be a contestant in the Jeopardy! quiz show, the DeepQA system had multiple information sources. The DeepQA system focused on extracting and scoring evidence from unstructured data, although it also used structured an semantic data sources. It was build as a massively probabilistic evidence-based architecture using more than 100 different techniques for analysing natural language. It used Apache UIMA[5], an Unstructured Information Management system.

2.4 Linked Data Systems

Linked Data consists in a set of rules about publishing Semantic Data in the web so it can be interlinked and accessed using semantic queries. The term was first used by Tim Berners-Lee while talking about the Semantic Web project. Additionally, Linked Open Data is an extension of the Linked Data concept, requiring that the data provided is open content.

There are a number of Linked Data Systems publicly available to the public, like DB-pedia[6], as long as multiple projects allowing to deploy your own linked data services. Some of them are described in the next sections.

2.4.1 Apache Lucene and Solr

Not considered Linked Data Systems on its own, Apache Lucene[7] is a high-performance, full-featured text search engine, that can be used as the foundation of many systems. Apache

[5]http://uima.apache.org/
[6]http://wiki.dbpedia.org/
[7]http://lucene.apache.org/

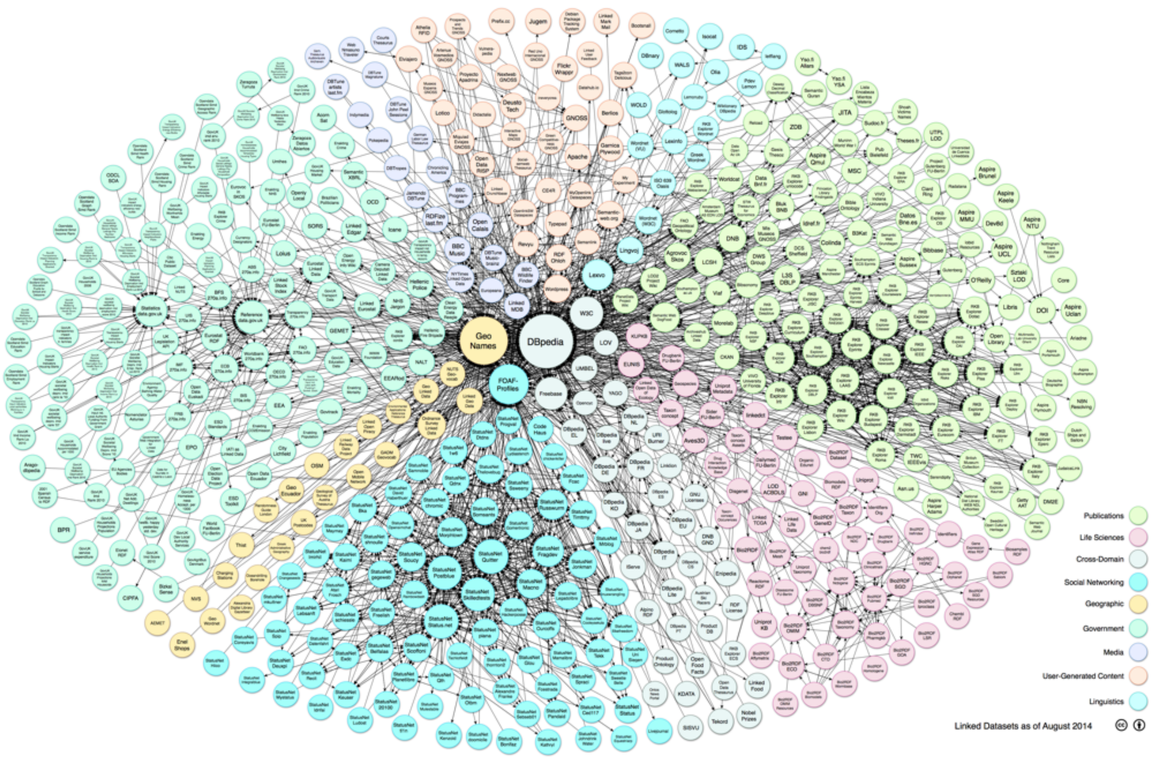


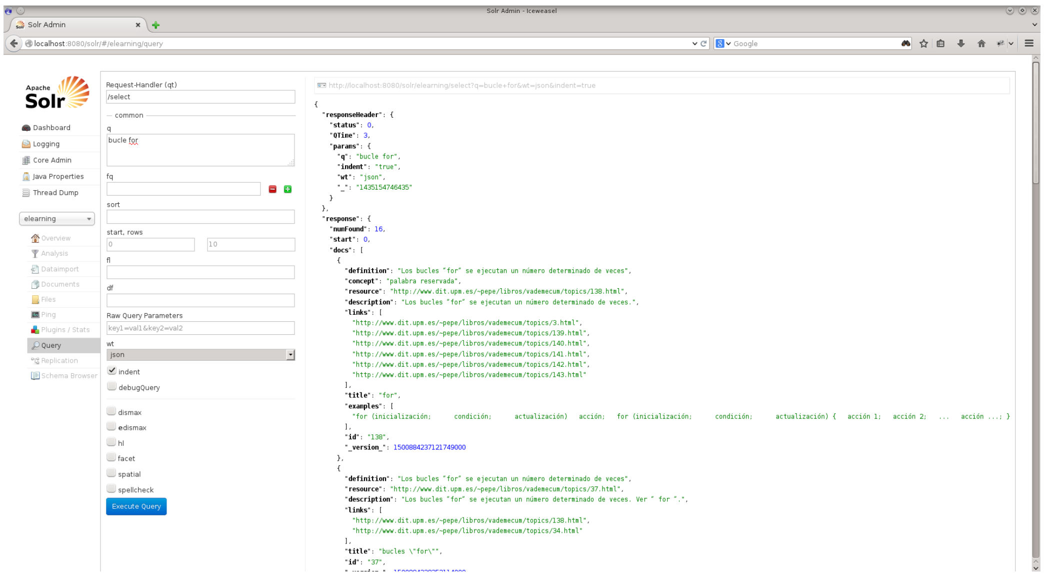
Figure 2.2: Linked Open Data cloud[a]

[a]http://lod-cloud.net/

Solr is built on top of Lucene, providing distributed indexing, replication and querying, with multiple features and functionalities[8]:

- Full-text search, with powerful matching capabilities, powered by Lucene.
- Optimized for High Volume traffic.
- Standards Based Open interfaces, including JSON, XML and HTTP.
- Comprehensive Administration Interfaces, making it easy to handle Solr instances.
- Easy Monitoring, publishing relevant data via JMX
- Highly scalable and Fault tolerant, using Apache ZooKeeper, is easy to scale and distribute the load.

Apache Solr is used as a base in many other systems, including some of the described in the following sections, and can also be used as a QA system of its own [16].



Request-Handler (qt): /select
common
q: bucle for
fq:
sort:
start, rows: 0 10
fl:
df:
Raw Query Parameters: key1=val1&key2=val2
wt: json
[✓] indent
[ ] debugQuery
[ ] dismax
[ ] edismax
[ ] hl
[ ] facet
[ ] spatial
[ ] spellcheck

http://localhost:8080/solr/elearning/select?q=bucle+for&wt=json&indent=true

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 3,
    "params": {
      "q": "bucle for",
      "indent": "true",
      "wt": "json",
      "_": "1435154746435"
    }
  },
  "response": {
    "numFound": 16,
    "start": 0,
    "docs": [
      {
        "definition": "Los bucles “for” se ejecutan un número determinado de veces",
        "concept": "palabra reservada",
        "resource": "http://www.dit.upm.es/~pepe/libros/vademecum/topics/138.html",
        "description": "Los bucles “for” se ejecutan un número determinado de veces.",
        "links": [
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/3.html",
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/139.html",
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/140.html",
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/141.html",
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/142.html",
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/143.html"
        ],
        "title": "for",
        "examples": [
          "for (inicialización;      condición;      actualización)   acción;   for (inicialización;      condición;      actualización) {   acción 1;   acción 2;   ...   acción ...; }
        ],
        "id": "138",
        "_version_": 1500884237121749000
      },
      {
        "definition": "Los bucles “for” se ejecutan un número determinado de veces",
        "resource": "http://www.dit.upm.es/~pepe/libros/vademecum/topics/37.html",
        "description": "Los bucles “for” se ejecutan un número determinado de veces. Ver “ for ”.",
        "links": [
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/138.html",
          "http://www.dit.upm.es/~pepe/libros/vademecum/topics/34.html"
        ],
        "title": "bucles \"for\"",
        "id": "37",
```

Figure 2.3: Web interface for Solr queries.

2.4.2 Linked Media Framework and Apache Marmotta

Started as the Linked Media Framework[9], this project aimed to provide an easy to setup server to offer linked media management, publishing Linked Data and allowing interactions

[8] http://lucene.apache.org/solr/
[9] https://bitbucket.org/srfgkmt/lmf/

with it. Linked Media Framework (LMF) is built in modules, some of them optional, allowing the extension of the functionalities in the Linked Media Server. Some of the implemented modules are:

- LMF Semantic Search allowing for a search service on top of Apache Solr.
- LMF Stanbol Integration, using Apache Stanbol for content analysis and interlinking.
- LMF SKOS editor, to manage SKOS thesauruses imported in the Linked Media Server.

The core functionalities of LMF were set aside to incubate Apache Marmotta, an Open Platform for Linked Data within the Apache Software Foundation[10], aiming to provide an implementation for Linked Data that can be used to both publish Linked Data and build custom applications for Linked Data.

2.4.3 Fuseki and Apache Jena

Fuseki, built using Apache Jena[11], is a SPARQL Protocol and RDF Query Language (SPARQL) server that provides a REST-style interface for SPARQL queries. It is built using Apache Jena, an open source Semantic Web framework for Java. Apache Jena provides an API to extract data and write it to RDF graphs, that are represented as an abstract model. A model can be sourced with data from files, databases, URLs or any combination of them. It also provides support for Web Ontology Language (OWL), and comes with several internal reasoners, as well as having support to use the Pellet[12] reasoner for OWL

2.5 Information retrieval

The amount of information available in the web has grown exponentially over the last years, with standards such as Linked Data helping exchange data among heterogeneous systems. However, many times these standards are not followed, and so it becomes necessary to recover and convert the data into compatible formats. There are multiple frameworks capable of crawling the web and recovering the relevant pieces of information, but we will focus here in Scrappy, a framework in ruby that allows extracting information from web pages and producing RDF data, and Scrapy, a Python tool that allows extracting data from websites into any format, with powerful capabilities.

[10]http://www.apache.org/
[11]http://jena.apache.org
[12]https://www.w3.org/2001/sw/wiki/Pellet

2.5.1 Scrappy

Scrappy [17] is a ruby framework with multiple functionalities, including web and REST interface, storing the scrapped data in a RDF repository, and outputting it in multiple formats. For our system, we will discuss the web scrapping and RDF output capabilities.

To recover data from a web page, Scrappy utilizes an ontology[13] to define the mapping between the web data and the Semantic Web resources. An example extractor can be found in listing 2.3 will take the link to the GSI's staff page and return a FOAF object with the name of each member. As seen in the extractor, it matches CSS elements in the page to semantic properties. Although not shown in this particular example, it can also use XPath and regular expressions as selectors.

```
dc: http://purl.org/dc/elements/1.1/
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
sioc: http://rdfs.org/sioc/ns#
sc: http://www.gsi.dit.upm.es/ontologies/scraping.rdf#

_:gsipeople:
  rdf:type: sc:Fragment
  sc:type: foaf:Person
  sc:selector:
    *:
      rdf:type: sc:UriPatternSelector
      rdf:value: "http://www.gsi.dit.upm.es/index.php?option=com_jresearch&
          view=staff&layout=positions"
  sc:identifier:
    *:
      rdf:type: sc:BaseUriSelector
  sc:subfragment:
    *:
      sc:type: foaf:Person
      sc:selector:
        *:
          rdf:type: sc:CssSelector
          rdf:value: ".jrperson"
      sc:identifier:
        *:
          rdf:type: sc:CssSelector
          rdf:value: "a"
          sc:attribute: "href"
      sc:subfragment:
```

[13] http://www.gsi.dit.upm.es/ontologies/scraping/

```
        *:
          sc:type: rdf:Literal
          sc:relation: foaf:givenName
          sc:selector:
            *:
              rdf:type: sc:CssSelector
              rdf:value: "a"
```

Listing 2.3: Example extractor for scrappy

An example of data generated with that extractor is shown in listing 2.4. It can be seen how the elements matched in the extractor have been converted into properties.

```
<rdf:RDF
  xmlns:foaf="http://xmlns.com/foaf/spec/#term_"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:sioc="http://rdfs.org/sioc/ns#">

 <foaf:Person rdf:about="http://www.gsi.dit.upm.es/index.php?option=
     com_jresearch&amp;view=member&amp;task=show&amp;id=84">
   <foaf:givenName>Alberto Mardomingo Mardomingo</foaf:givenName>
 </foaf:Person>
 <foaf:Person rdf:about="http://www.gsi.dit.upm.es/index.php?option=
     com_jresearch&amp;view=member&amp;task=show&amp;id=16">
   <foaf:givenName>Carlos A. Iglesias</foaf:givenName>
 </foaf:Person>
</rdf:RDF>
```

Listing 2.4: Example person extracted with Scrappy

Using the appropriate selectors, scrappy can follow the links on a page, automating the scraping of big web sites, and converting all the data into RDF, N-Triples or JSON-LD.

2.5.2 Scrapy

Scrapy[14] is a fast high-level web crawling framework, used to extract structured data from websites. It is written in Python, and by default outputs data to JSON, although it accepts custom exporters, giving the user the ability to export into any format it requires.

[14]http://scrapy.org/

The crawlers are Python classes that extend the Spider class in scrapy.

```
import scrapy
from vademecum.items import VademecumItem

import re
import bs4
import json

class vademecumSpider(scrapy.Spider):

  # Test spider to get the vademecum.IO
  name = "vademecum"
  allowed_domains = ["http://www.dit.upm.es/"]
  url_base = "http://www.dit.upm.es/~pepe/libros/vademecum/topics/{}.html"

  start_urls = [url_base.format(str(i)) for i in xrange(4, 394)]

  def parse(self, response):

    doc = VademecumItem()
    doc["resource"] = response.url

    header = response.xpath(
          "//p[@class=\"MsoHeader\"]/ancestor::div/*[2]/descendant::*/*/
              text()")
          .extract()
    header = ''.join(header).strip()
    match = re.search("(\d+)\.\s*(.+)\s\[(.+)\]\s?\((.+)\)",
                      header, flags=re.U)
    if match and match.lastindex == 4:
      doc["title"] = match.group(2)
      doc["alternative"] = match.group(3)
      doc["concept"] = match.group(4)

    return doc
```

Listing 2.5: Example scrapy spider

Listing 2.5 shows part of a scrapy spider that will return a JSON object containing the URL of the scrapped page, as well as the title, alternative and concept fields scrapped from the document.

2.6 Web technologies

Known simply as “The web”, the World Wide Web is an information system where hypertext documents are accessed via the internet. First proposed by Tim Berners-Lee in 1989 [18], it has grown to be used by two out of five people around the world[15].

The technologies used in web services can be divided in Client technologies, executed in the user’s computer, and Server technologies, executed in the server side of the service. We will provide a short description of some of the technologies available for each side, focussing on those used in this project.

2.6.1 Client technologies

Web browsers are usually responsible for running the code of a website. Usually, that code consists on CSS, HTML and JavaScript files, that are interpreted by the browser to present the page, and respond to the user actions.

- **HTML**: or *HyperText Markup Language* is the standard markup language used to create web pages. It consists on a collection of pairs of tags that identify the different elements on a page, describing the structure of the page. It can also include images and other objects, allowing for complex user interaction.
- **CSS**: for *Cascading Style Sheets* is a style sheet language, used mostly to describe the look and formatting of documents written in a markup language such as HTML. It is designed to allow separation of content from document presentation, providing more flexibility and control of the presentation, while also improving accessibility.
- **JavaScript** is a programming language used to run scripts to interact with the user inside the browser window. Along with JavaScript, its most used library[16] is jQuery, which is designed to simplify many of the usual tasks performed with JavaScript.

These technologies are often used with Ajax (*Asynchronous Javascript XML*), a group of web development techniques used to create asynchronous web applications. Using Ajax, web applications can interact with the server in the background, therefore not interfering with the behaviour and graphical display of the rest of the page.

[15]`http://webfoundation.org/about/vision/history-of-the-web/`
[16]`http://libscore.com/#libs`

2.6.2 Server technologies

The interactions presented by the web client are the processed in the server side, usually communicating using HTTP. There are multiple applications capable of handling this interaction, known as HTTP servers. Apache, NginX or Microsoft Windows Server® are some of the most popular servers[17].

For our service, we have used Apache [19], an Open Source HTTP-Server. First launched in 1995, it has continued development to this date, with version 2.4.12 being released on January 2015[18]. It is currently developed and maintained by an open community of developers under the Apache Software Foundation, and made available in a wide variety of operating systems, including GNU/Linux and Microsoft Windows®. It features a module-based system, allowing the core functionality to be expanded by compiled modules. Some of the most popular modules include:

- **mod_php** Enabling the use PHP to execute server side code, this module can be found in many Apache installations, allowing the deployment of services like WordPress or Joomla.
- **mod_auth_basic** Handling basic user authentication, this module allows the server administrator to block sections of the server from being accessed by the general public.
- **mod_proxy** Allows the use of Apache as a proxy, masking other services behind it.

Over the last few years, there has been an important trend in the use of evented web technologies, a vision of the traditional web APIs complemented with other APIs that produce events, and provide a callback mechanism, making the Web more like a giant network. Node.js[19] is one of the most popular environments to build this kind of applications.

2.6.2.1 WSGI Servers in Python

Web Server Gateway Interface (WSGI) is a specification for simple interfaces between web servers and web applications for the Python programming language. First defined in PEP 333 [20], and updated in PEP 3333 [21], it has been adopted as a standard for Python web application development.

[17]`http://news.netcraft.com/archives/2015/05/19/may-2015-web-server-survey.html`
[18]`http://httpd.apache.org/`
[19]`https://nodejs.org/`

There are multiple implementations and frameworks of WSGI for Python, some of the most popular are:

- **Bottle** is a simple lightweight WSGI framework, focusing on simplicity. It is distributed as a single-file module, with no other dependencies than the Python standard library. However, it has capabilities to handle routing, easy access to web data such as cookies and HTTP headers, and includes a built-in server for development. It also has support for templates, both with a built-in engine, and using external modules such as mako or jinja2.
- **Django** is a full-fledged Python web framework, offering fast and scalable services, with multiple built-in options, such as security and administration tools. It is slightly higher level than other frameworks, and emphasizes reusability of components and plugins, as well as rapid development.
- **Flask**, a micro web application framework, based on the jinja2 template engine, and the Werkzeug WSGI toolkit. It focuses on providing a simple interface, whilst still providing multiple features such as RESTfull request dispatching, cookies and request handling and unicode support. It also includes native support for unit testing, as well as a development server and debugger. It supports extensions for extra functionalities.

In our application, we have chose Apache as the gateway server, using mod_wsgi, and Flask for the application itself.

2.7 Summary

In this section, we have discussed the technologies related to the system we are developing.

First, we took a look at *conversational agent* systems. We saw that AIML and its implementations provide a robust technology, that is still evolving despite its age. It has been widely used, and still has an large community and volume of users, and is working on the new AIML 2.0. We also considered ChatScript characteristics, with better performance and a new language for writing bots, with a simpler syntax and features not present in AIML 1.

We then considered *Question Answering* systems and its different approaches, also taking a look at some systems that use each approach.

As for *Linked Data Systems*, we studied the concept of the Semantic Web and Linked Data, to then take a look at multiple systems used for it: Apache Solr, not a linked data system on its own, but used in many of them, the Linked Media Framework, a full-fledged Linked Data server built around Apache Marmotta, and Fuseki, a SPARQL server with reasoning capabilities built with Apache Jena.

After considering the data indexing systems, we studied two *Information Retrieval* tools: Scrappy, a ruby web crawler capable of exporting RDF documents from the data extracted from the web, and Scrapy, a Python framework for web spiders, with support for multiple functionalities as well as allowing the export of the data in personalised formats.

Finally, we took a quick look at *web technologies*, both for client-side applications, as well as server side systems that would allow complex interactions with the user. We focussed specially on the server side, studying different implementations of WSGI modules for Python, and considering their characteristics.

CHAPTER 3

Architecture

In this section, we describe the overall architecture of a Question Answering system that features social dialogue in a learning environment. First, we will summarize the requirements for the system, and discuss the global architecture, to then explain in detail each of the modules for the system. Finally, we will follow the process of two use cases, and describe the process for each of them.

3.1 Overview of the modules

In this section we will describe the architecture of our system, starting with the modules identified in the requirements. These requirements are:

(i) Present users with a simple interface for their questions, and to present them with the answers.

(ii) Classify each input and handle it to the appropriate module.

(iii) Track dialogue options according to the speech.

(iv) Implement a QA system that would be able to search the document library and extract a short answer.

(v) Extend the documents library, scrapping external documents and producing semi-structured indexes.

(vi) Follow up the learning process, and use it to improve the learning experience.

In figure 3.1 we show the global architecture of the system, identifying the main modules: Conversational Agent, Question Answering and Information Extraction Agent. In the rest of this section we will discuss the function of each module.

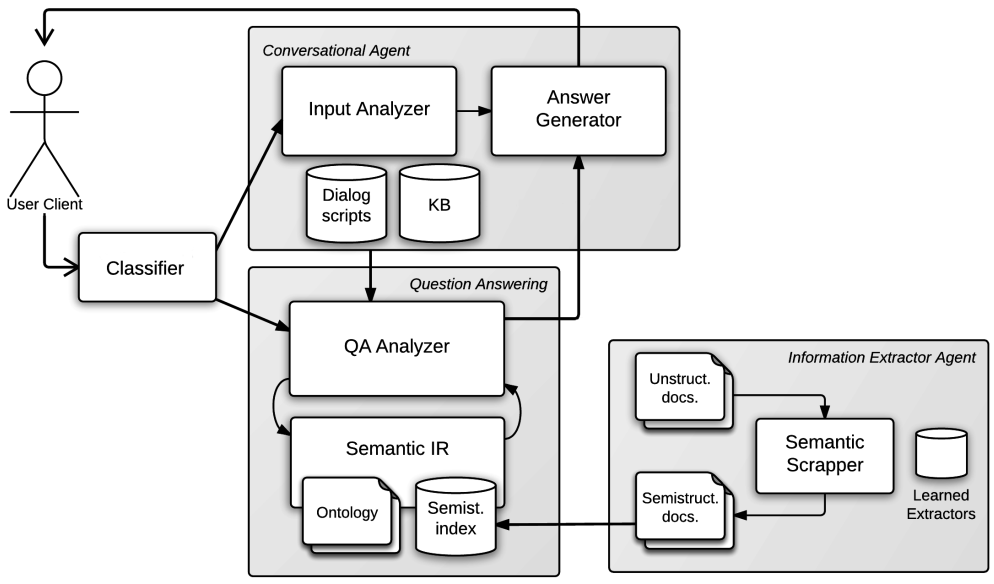


Figure 3.1: Global view of the architecture proposed

The classifier receives the user queries. It provides an endpoint that the user-agent (or different user-agents) uses to post the requests and obtain responses (i.e. the system interface). The classifier then routes the user query to the conversational agent or the QA modules depending on how the input is classified. Nevertheless, the architecture may be extended with additional modules for multiple types of input. This classifier is usually implemented using a supervised machine learning approach with algorithms such as decision trees or naive Bayes that provide decent results for this purpose [22, 23]. The classifier carries out a preliminary analysis of the query, since other modules will perform a deeper analysis of the input depending on the kind of input they handle.

3.1.1 Conversational Agent

The *Conversational Agent* is responsible for handling the social dialogue of the conversation (also referred by other authors as small talk or chit chat). It also traces the topic of the conversation that stores in the fact KB, along with the former utterances and pills of information learned or devised from the user. It is responsible for understanding the whole meaning of the user query when he/she omits information already provided in previous utterances. The Input Analyser inside the Conversational Agent, performs a script based analysis of the input by matching regular expressions against the input. Some advanced implementations of script based analysers also use dictionaries and perform Part of Speech (PoS) tagging and parsing.

The Conversational Agent, by means of the Answer Generator, is also in change of generating the answers that will be sent to the users. These are also stored in the dialogue scripts. In the most common scenario, the small talk input is analysed by the Input Analyser and tells the Answer Generator to given a particular answer in response. However, the QA can also instruct the Answer Generator to send a response to the user. In that case, the Conversational Agent generates the answer according to the topic and former utterances, giving it a particular touch when needed. Besides the textual response that is presented to the user, the answer is decorated with meta-data to provide further information about the state of the Conversational Agent. Typically, they indicate the mood of the agent, its facial expressions and gestures (possible implemented using Behavior Markup Language (BML)), etc.

3.1.2 Question Answering

The *Question Answering* takes part with those user queries where he/she asks for a particular piece of information, i.e. as with a regular QA system. The QA Analyser uses domain-specific grammars to extract the precise meaning of the query. This is, it classifies the type of query, extracts the relevant concepts, and categorize them according to the ontologies considered. The effectiveness of the process depends of the accuracy and degree of detail of the grammars applied, which includes the precision of the concept categorization. The QA Analyser combines general-scope dictionaries with domain-specific ones to enhance its effectiveness. If there is no grammar that can be applied to the query, the analysis simply does not return any outcome.

As with regular QA systems, the Information Retrieval (IR) is consulted to obtain the relevant documents where the answer is contained. The IR works with a set of documents that has been previously indexed. In this case, given the semantic nature of the IR system, it also acts as a SPARQL endpoint that may be queried for precise pieces of information. Thus, it supports a dual working mode depending on the nature of the query. With semantic queries the results are more precise and the information returned has better structure, the better categorization of the fields returns the more accurate results. Semantic queries also enable the use of linked data not only for enhancing results but for query expansion. When the semantic IR is not available –either because the incoming query is too general, or because there is no relevant structured documents–, the IR module will do its best to return a piece of information as accurate as possible. At least, it retrieves a set of documents that are related to the query. It will try to categorize the nature of the documents, which brings the category of the concept searched, that can be used to expand the query and reformulate the query. If no relevant document is returned by the IR, the QA is not capable to give a response, and thus the Answer Generator will inform the user.

Moreover, the Semantic IR may also be queried by external modules. The treatment given to the query is the same as when it comes from the QA Analyser. Finally, the conversational agent may derive a query to the QA in those cases where the Classifier miss-classifies the query, and more frequently with those utterances where the user asks for more information. In this case, the QA system needs further information to perform the document retrieval. Thus, the conversational agent will expand the query and route it to the QA.

3.1.3 Information Extractor

In case there is no relevant document for the query performed, the system will try update the KB. The *Information Extractor Agent*'s main function consists on analysing unstructured documents in order to extract fields, categorize them and generating a semi-structured document.

This is a slow process, so it cannot be performed in near real-time; instead, unresolved query may trigger its execution, that will be available for future queries. This automatic information extraction mechanism is a best-effort process that relies on the information on the sources, and the ontologies used to map that information. Semantic scrappers such as Scrappy [17] may boost this process. Alternatively, a system administrator may also manually include documents on the IR index, but also mark them to be processed by the Information Extractor Agent and index them afterwards.

3.2 Work process

In this section we will describe the process a question introduced into the system will follow. Depending on the user input, the aforementioned process may differ. Here, we will consider two types of user input, the first one being a simple social dialogue sentence, that won't require a look up in the KB. The other type of input to be considered is an actual question, requiring a lookup in the KB.

The classifier will differentiate between the distinct types of input, and trigger the appropriate processes.

3.2.1 Simple sentence

Our system is ready to handle social chat, which aims to provide a richer experience to the user, encouraging him to keep chatting and having a better learning experience. In this case, a KB lookup is not required, and therefore, the process is as stated:

1. The user inputs the sentence into the system.
2. The classifier tags the input as social dialogue.
3. The input analyser decides which type of social interaction we are facing, such as a greeting, a identifying question ("Are you the teacher? ") or an insult, among others.

4. The answer generator provides an appropriate response and returns it to the user.

This process is shown graphically in Figure 3.2.

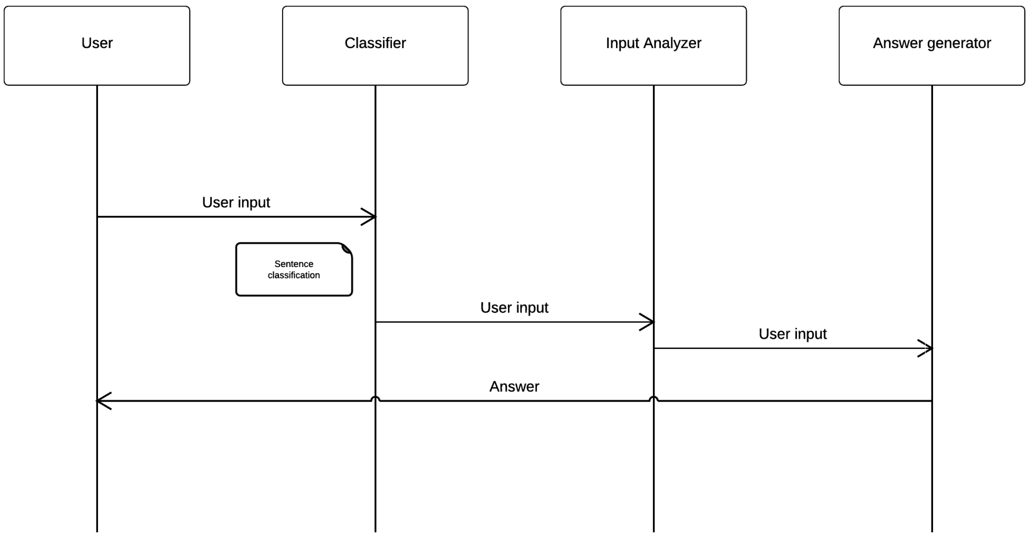


Figure 3.2: Simple sentence process

3.2.2 Question with KB lookup

In this process, the question input into the system does require a lookup in the KB, via the QA system. The process to produce the output will be the following:

1. The user inputs the question into the system.
2. The classifier tags the input as an actual question.
3. The QA Analyser process the question and performs a search in the KB.
4. The QA Analyser returns the response for the question to the Answer generator.
5. The answer generator process the response and returns a natural question, as well as the required Out of Band (OoB) command with the data.

Figure 3.3 shows the process graphically.

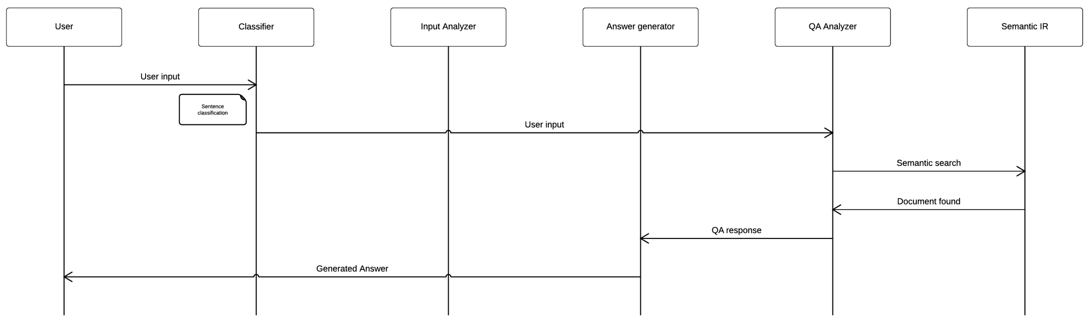


Figure 3.3: Question with KB lookup process

3.3 Summary

In this chapter, we presented the proposed architecture for our system.

We started by taking a look at the *requirements* that any proposed system should accomplish, and taking a look at the module structure we propose, as well as a short description for the function of each module.

We then proceeded to take a look at each module in detail, analysing its functionality, as well as how each module interacts with each other.

Finally, we study two possible use cases for the systems, the first one without requiring a look up on the Knowledge Base, an the second requiring a full lookup to answer the question and supply with a relevant document for the user.

CHAPTER 4

Case study: Java elearning platforms

In this chapter, we will describe the prototype we developed for a Java Bot following the architecture described in the previous chapter. We will start with a short overview of the implemented modules for the system, to then take a detailed look to each one of them, describing how they work, and how they connect to each other.

4.1 Overview of the system

For this system, we developed a prototype utilizing the architecture explained in chapter 3. To do so, we deployed the following modules and subsystems:

1. A Javascript client, that will connect to the system and act as a user interface.
2. A Front-end controller, written in Python, that will handle the interaction between the different modules.
3. A chatbot using ChatScript, to handle question analysis and social dialogue.
4. An Apache Solr instance, where all the semantic data will be loaded.

In parallel to all this we developed both a web scraper using scrapy to recover the relevant data, and an uploader to post the data to Solr.

4.2 Overall process

In this section we will provide a short explanation on how each module of the system interact with each other. Figure 4.1 shows a schematic view of the process the system follows, first for the retrieval of the information and, when the system if fully deployed, facing a query from a user.

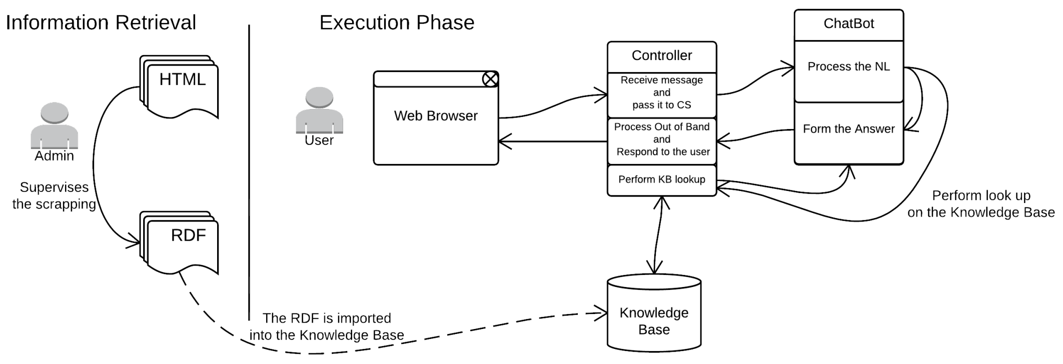


Figure 4.1: Overall cycle for the system.

First, the data from the Vademecum need to be scrapped into RDF documents, in an automatic process that can be supervised by a human administrator, but can also be fully automated. More details on this process can be found on section 4.3. The generated RDF data is then stored into our Knowledge Base, so it can be easily queried. The process of scrapping the data and indexing it can be performed periodically to include new data as it appears.

Once the system is deployed, users can input questions using the interface explained in section 4.4. Questions from the users will be packaged in a JSON request and then sent to the controller, that will process the query.

The controller, described in section 4.5, will start the process sending the question to the Chatbot (section 4.6). The Chatbot will process the Natural Language input from the user, and classify it, attending on whether the question is simple social dialogue or requires a knowledge base lookup. In the former case, the Chatbot will simply generate a Natural Language response and send it to the controller, which will process it and sent it back to the chat client and the user.

For the questions requiring Knowledge Base lookup, the Chatbot will identify the topic of the question and will send an Out of Band command to the controller, asking for the lookup. The controller will then form the query to retrieve the requested data, and perform the query. With the retrieved information, the controller will form a new out of band command, and send it back to the Chatbot. It will then process the data, and form a natural language response, sending it back to the controller for its final processing before being returned to the user in the chat client. The detailed information about the Knowledge Base can be found in section 4.7.

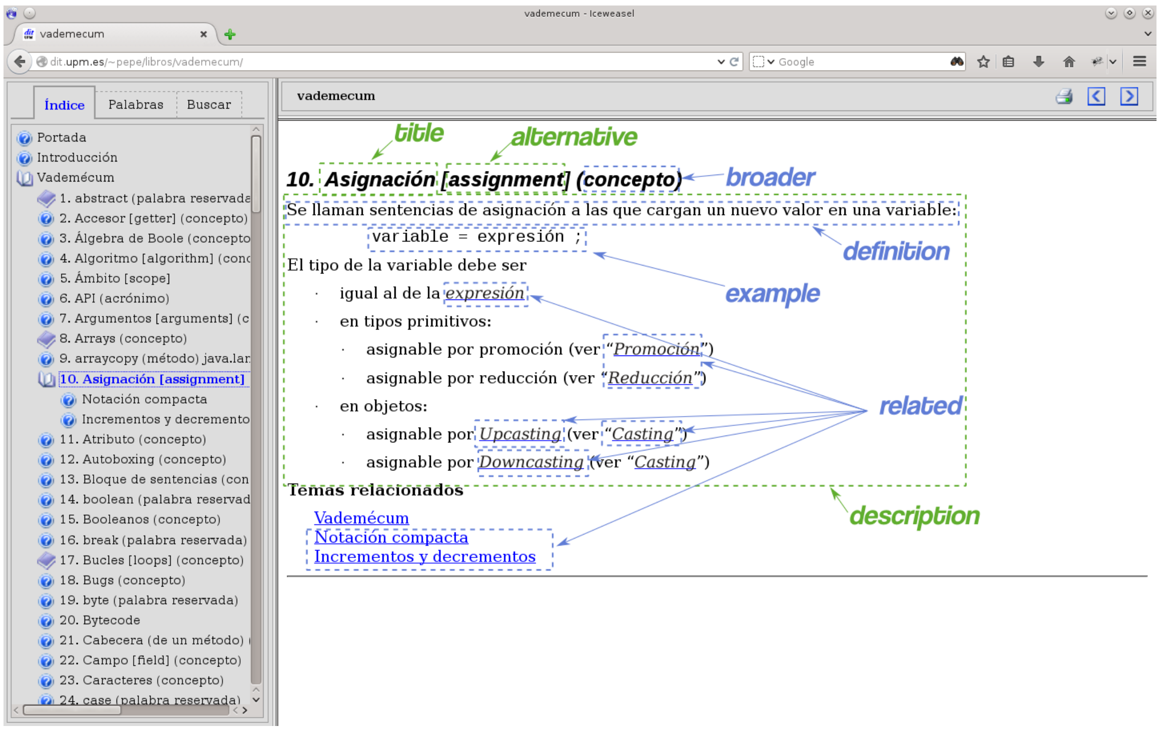
4.3 Scrapping process

The data for this prototype comes from Professor Jose A. Mañas' Java Vademecum[1], converted into RDF and JSON format using scrapy to be uploaded to a Solr instance. In this process, we will study the structure of a Vademecum document and match it to the schema described in section 4.7.1. An example of that conversion is shown in figure 4.2.

The process for that document is automated using Python, XPath and regular expressions, using jinja2[2] templates to generate the data. The scrapy spider will go through every entry in the Vademecum and create a list of items, which will in turn be exported in the required format. We have considered two formats for the export, JSON and RDF. Both of them are based on a taxonomy using SKOS and Dublin Core. However, it is worth noting

[1] `http://dit.upm.es/~pepe/libros/vademecum/`

[2] `jinja.pocoo.org`



Portada
Introducción
Vademécum
1. abstract (palabra reservada
2. Accesor [getter] (concepto)
3. Álgebra de Boole (concepto
4. Algoritmo [algorithm] (conc
5. Ámbito [scope]
6. API (acrónimo)
7. Argumentos [arguments] (c
8. Arrays (concepto)
9. arraycopy (método) java.lar
10. Asignación [assignment]
Notación compacta
Incrementos y decremento
11. Atributo (concepto)
12. Autoboxing (concepto)
13. Bloque de sentencias (con
14. boolean (palabra reservad
15. Booleanos (concepto)
16. break (palabra reservada)
17. Bucles [loops] (concepto)
18. Bugs (concepto)
19. byte (palabra reservada)
20. Bytecode
21. Cabecera (de un método)
22. Campo [field] (concepto)
23. Caracteres (concepto)
24. case (palabra reservada)

10. Asignación [assignment] (concepto)

Se llaman sentencias de asignación a las que cargan un nuevo valor en una variable:

```
variable = expresión ;
```

El tipo de la variable debe ser

- igual al de la *expresión*
- en tipos primitivos:
  - asignable por promoción (ver "*Promoción*")
  - asignable por reducción (ver "*Reducción*")
- en objetos:
  - asignable por *Upcasting* (ver "*Casting*")
  - asignable por *Downcasting* (ver "*Casting*")

Temas relacionados

Vademécum
Notación compacta
Incrementos y decrementos

Figure 4.2: Example mapping for the vademecum.

that we are using JSON rather than JSON-LD, so we also provide the RDF as Semanic Data.

4.3.1 RDF for Java

The first part of the process is to define a taxonomy for the Java elements in the Vademecum. We have chosen to do this using SKOS[3] and Dublin Core[4], both of them used for the representation of documents as Linked Data. We have therefore modelled a single Vademecum page with the fields shown in table 4.1. Along with the objects representing the Vademecum documents, our scrapper will generate a SKOS ConceptScheme, with all the concepts identified in the Vademecum.

The data will be exported in RDF/XML format. An example of an object can be shown in listing 4.1. It's worth noting that the URL for the described document is present as the rdf:about field.

[3]http://www.w3.org/TR/skos-reference/
[4]http://dublincore.org/documents/dcmi-terms/

Parameter	Source	Description
title	Dublin Core	The name of the document, in Spanish.
alternative	Dublin Core	The English equivalent for the title.
definition	SKOS	A short description of the term.
description	Dublin Core	The full text of the document.
exmaple	SKOS	An example of the element described.
related	SKOS	A topic related to the one being described.
inScheme	SKOS	The SKOS concept scheme this element is part of.
broader	SKOS	The parent SKOS concept for this element.

Table 4.1: Fields used representing a Java Object.

```
<skos:Concept rdf:about="http://www.dit.upm.es/~pepe/libros/vademecum/
    topics/26.html">
    <dcterms:title xml:lang="es">Asignacion</dcterms:title>
    <dcterms:alternative xml:lang="en">assignment</dcterms:alternative>
    <skos:definition>Se llaman sentencias de asignacion a las que cargan
        un nuevo valor en una variable</skos:definition>
    <dcterms:description>Se llaman sentencias de asignacion a las que
        cargan un nuevo valor en una variable: El tipo de la variable
        debe ser igual al de la expresion en tipos primitivos: asignable
        por promocion (ver \" Promocion\") asignable por reduccion (ver
        \" Reduccion\") en objetos: asignable por Upcasting (ver \"
        Casting\") asignable por Downcasting (ver \"Casting\")</dcterms:
        description>
    <skos:related>http://www.dit.upm.es/~pepe/libros/vademecum/topics
        /122.html</skos:related> <skos:related>http://www.dit.upm.es/~
        pepe/libros/vademecum/topics/242.html</skos:related> <skos:
        related>http://www.dit.upm.es/~pepe/libros/vademecum/topics/247.
        html</skos:related> <skos:related>http://www.dit.upm.es/~pepe/
        libros/vademecum/topics/288.html</skos:related> <skos:related>
        http://www.dit.upm.es/~pepe/libros/vademecum/topics/47.html</skos
        :related> <skos:related>http://www.dit.upm.es/~pepe/libros/
        vademecum/topics/86.html</skos:related> <skos:related>http://www.
        dit.upm.es/~pepe/libros/vademecum/topics/47.html</skos:related> <
        skos:related>http://www.dit.upm.es/~pepe/libros/vademecum/topics
        /3.html</skos:related> <skos:related>http://www.dit.upm.es/~pepe/
```

Json field	RDF equivalent	Description
title	dcterms:title	The name of the document, in Spanish.
alternative	dcterms:alternative	The English equivalent for the title.
definition	skos:definition	A short description of the term.
description	dcterms:description	The full text of the document.
example	skos:example	An example of the element described.
links	skos:related	A topic related to the one being described.
id		An unique identifier for this document, matching the html document.
concept	skos:broader	The corresponding concept in the concept scheme.

Table 4.2: Json fields and its equivalence.

```
            libros/vademecum/topics/27.html</skos:related> <skos:related>http
            ://www.dit.upm.es/~pepe/libros/vademecum/topics/28.html</skos:
            related>
        <skos:example>variable = expresion ;</skos:example>
        <skos:broader>concepto</skos:broader>
        <skos:inScheme>VademecumScheme</skos:inScheme>
    </skos:Concept>
```

Listing 4.1: Example extractor

4.3.2 JSON data

Aside from the generated RDF, the scrapper has also de ability to generate JSON. However, this format comes with some limitations, such as not including the concept scheme, and not keeping the source of the fields. As can be seen in listing 4.2, a single JSON document will take the URL as identifier, and have similar fields than the RDF. The correspondence between the RDF and the JSON fields is shown in table 4.2

This format will be used to upload the data about the Vademecum documents to Solr.

```
"http://www.dit.upm.es/~pepe/libros/vademecum/topics/26.html": {
        "definition": "Se llaman sentencias de asignacion a las que cargan un nuevo valor en una variable",
        "concept": "concepto",
        "description": "Se llaman sentencias de asignacion a las que cargan un nuevo valor en una variable: El tipo
               de la variable debe ser igual al de la expresion en tipos primitivos:  asignable por promocion (ver
              \"Promocion\")  asignable por reduccion (ver \"Reduccion\")  en objetos:  asignable por Upcasting (ver
               \"Casting\") asignable por Downcasting (ver \"Casting\")",
        "links": [
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/122.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/242.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/247.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/288.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/47.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/86.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/47.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/3.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/27.html",
            "http://www.dit.upm.es/~pepe/libros/vademecum/topics/28.html"
        ],
        "example": "variable=expresion;"
        "title": "Asignacion",
        "alternative": "assignment",
        "id": "26"
    },
```

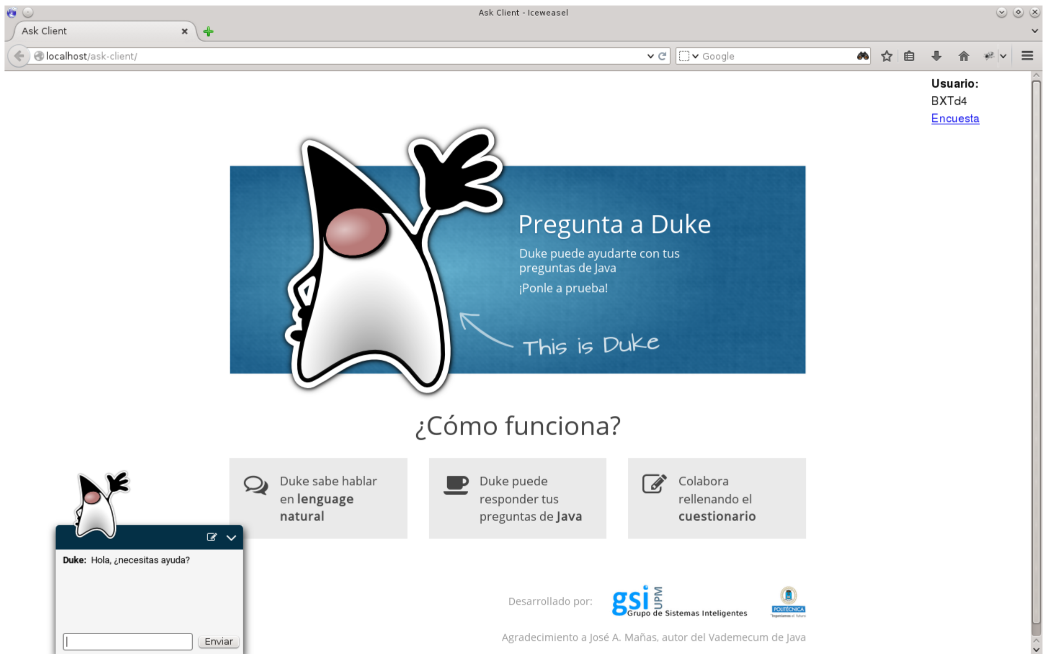
Listing 4.2: Example JSON document from the Vademecum

4.4 Chat client

In our prototype, the interaction with the system is done via a web client that provides a chat box and an iframe where the content is located. When the user first opens the page, it's greeted by the bot, and provided with a short explanation of how the client works.

The client is made using web technologies: HTML, CSS and Javascript, and uses Ajax to communicate with the server, sending the user questions and handling the response, waiting for the user to send a question and then making the request to the server, as shown in listing 4.3.

```
        $.ajax({ url: form.attr('action'),
                 data: json_data,
                 dataType: 'json',
                 contentType: 'application/json;charset=UTF-8',
                 success: populateForm,
                 error: function(data_resp) {
                        $('#screen').append(constructDialogEntry('Duke',
                               'Vaya, parece que he tenido algún error
                                   conectando con el servidor... Whoops'))
                                   ;
```



Usuario:
BXTd4
Encuesta

Pregunta a Duke

Duke puede ayudarte con tus preguntas de Java

¡Ponle a prueba!

This is Duke

¿Cómo funciona?

Duke sabe hablar en lenguage natural

Duke puede responder tus preguntas de Java

Colabora rellenando el cuestionario

Duke: Hola, ¿necesitas ayuda?

Desarrollado por: gsi UPM Grupo de Sistemas Inteligentes

Agradecimiento a José A. Mañas, autor del Vademecum de Java

Figure 4.3: Web interface for the client.

```
                        console.log("Error connection to the controller");
                    return false;
                 }
        });
        return false;
    });

    function populateForm (data_resp) {
        console.log(data_resp);
        if (data_resp.answer) {
            data_resp.answer.forEach(function(answer) {
                $('#screen').append(constructDialogEntry('Duke', answer));
    scrollDisplay();
                if (data_resp.resource) {
                    if (data_resp.resource.indexOf('vademecum') != -1 ){
                        // This links to the vademecum
                        var name_start = data_resp.resource.lastIndexOf('/'
                            )+1
                        var filename = data_resp.resource.substring(
                            name_start)
                        $('#iframe-qa').attr('src', vademecum_base +
                            filename);
                        current_url_shown = vademecum_base + filename;
                    } else {
                        //I have some other link
                        $('#iframe-qa').attr('src', data_resp.resource);
                        current_url_shown = data_resp.resource;
                    }
                }
            });
        } else {
            $('#screen').append(constructDialogEntry('Duke',
                                'Lo siento, no puedo responder a esa
                                    pregunta'));
      scrollDisplay();
        }
        scrollDisplay();
    }
```

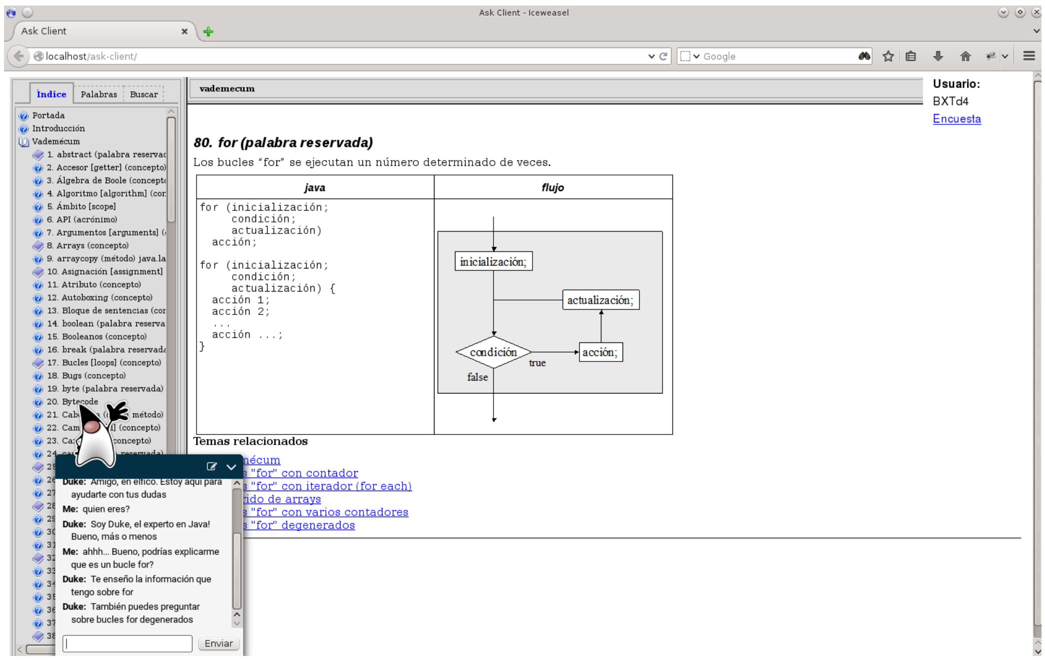
Listing 4.3: Ajax performing the request to the controller

Upon the user submitting a question through the interface, the client will send a GET request to the controller, as described in section 4.5, and will receive a JSON response, containing the answer and the page to be shown to the user, if existent. An example response to the question “¿Qué es un for?” is shown in listing 4.4.

```
{
  "answer": [
 "Esto es lo que s\u00e9 sobre for",
 "Si quieres, creo que bucles for degenerados tienen algo
    que ver con esto"
      ],
  "definition": "Los bucles for se ejecutan un n\u00famero
      determinado de veces",
  "links": [
             "http://www.dit.upm.es/~pepe/libros/vademecum
                /topics/3.html",
             "http://www.dit.upm.es/~pepe/libros/vademecum
                /topics/139.html",
             "http://www.dit.upm.es/~pepe/libros/vademecum
                /topics/140.html",
             "http://www.dit.upm.es/~pepe/libros/vademecum
                /topics/141.html",
                    "http://www.dit.upm.es/~pepe/libros/vademecum
                        /topics/142.html",
                    "http://www.dit.upm.es/~pepe/libros/vademecum
                        /topics/143.html"
                    ] ,
        "resource": "http://www.dit.upm.es/~pepe/libros/
            vademecum/topics/138.html"
    }
```

Listing 4.4: Example response for the chat client

Figure 4.4 shows the web interface after a few interactions with the user.



Ask Client - Iceweasel
Ask Client
localhost/ask-client/
Google
Índice
Palabras
Buscar
Portada
Introducción
Vademécum
1. abstract (palabra reservad
2. Accesor [getter] (concepto)
3. Álgebra de Boole (concept
4. Algoritmo [algorithm] (con
5. Ámbito [scope]
6. API (acrónimo)
7. Argumentos [arguments] (
8. Arrays (concepto)
9. arraycopy (método) java.la
10. Asignación [assignment]
11. Atributo (concepto)
12. Autoboxing (concepto)
13. Bloque de sentencias (con
14. boolean (palabra reserva
15. Booleanos (concepto)
16. break (palabra reservada
17. Bucles [loops] (concepto)
18. Bugs (concepto)
19. byte (palabra reservada)
20. Bytecode
vademecum
Usuario:
BXTd4
Encuesta
inicialización;
actualización;
condición
acción;
true
false
Enviar

80. for (palabra reservada)

Los bucles "for" se ejecutan un número determinado de veces.

java	flujo
`for (inicialización; condición; actualización) acción;` `for (inicialización; condición; actualización) { acción 1; acción 2; ... acción ...; }`	

```
for (inicialización;
     condición;
     actualización)
  acción;

for (inicialización;
     condición;
     actualización) {
  acción 1;
  acción 2;
  ...
  acción ...;
}
```

Temas relacionados

mécum
s "for" con contador
s "for" con iterador (for each)
rido de arrays
s "for" con varios contadores
s "for" degenerados

Duke: Amigo, en elfico. Estoy aqui para ayudarte con tus dudas
Me: quien eres?
Duke: Soy Duke, el experto en Java! Bueno, más o menos
Me: ahhh... Bueno, podrías explicarme que es un bucle for?
Duke: Te enseño la información que tengo sobre for
Duke: También puedes preguntar sobre bucles for degenerados

Figure 4.4: Web interface after a question.

4.5 Front end controller

The Front end controller is the main control module in our system. It handles the requests received from the client described in 4.4, and proceeds to triggers the required modules, as well as executing the Out of Band commands received from each module. This module is provided as a web service, and therefore we have chosen Flask [24] and Apache's mod_wsgi [25] to deploy it. In the following subsections we will describe how it works as well as its work-flow structure.

4.5.1 Functional Model

The function of this module is returning the answer to the user, formed as JSON, by triggering the appropriate modules and reacting to their responses. To do so, it follows a process explained in the UML diagram shown in Figure 4.5.

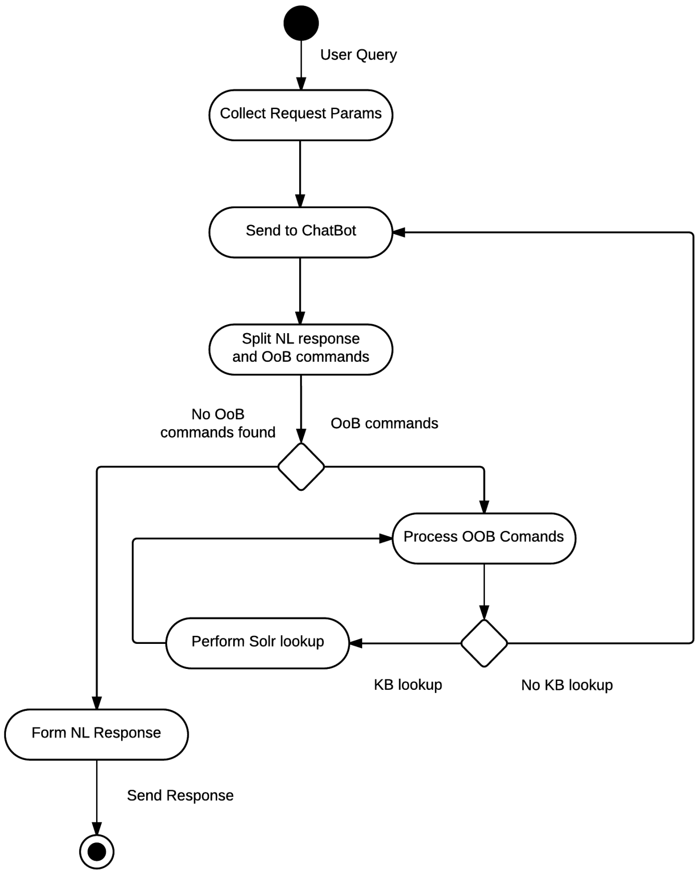


Figure 4.5: UML diagram of the process followed by the controller.

- **Request parsing:** The client sends the query as JSON using a HTTP request to the front-end controller. This JSON is recovered and processed into a Python dictionary.

- **Send to ChatBot:** The user query is sent to the ChatBot so it's processes and a response, either Natural Language (NL) or Out of Band, is generated.
- **Split NL response and Ouf of Band commands** The response in the previous step is split in NL and Out of Band commands, to process each one appropriately.
- **Out of Band command processing:** Read the Out of Band commands and take the appropriate steps for each one of them.
- **Solr Lookup:** If a lookup in the Knowledge base is required, send the query to Solr.
- **Form Response:** Once there are no more Out of Band commands, the controller forms the actual response in JSON and send it to the user.

4.5.2 Structural Model

In this section we will describe the structure followed designing the front end controller. In figure 4.6 we show the method structure of the controller.

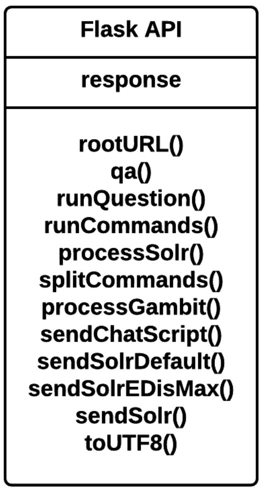


Figure 4.6: Front end controller structure

Now we will proceed to describe each of the methods shown in the figure.

- **rootURL()** will be triggered whenever the base URL for the controller is requested. It will act in the same way than the qa() method.

- **qa()** this function will recover the request parameters, shown in table 4.3, and start the process to understand the user query, in order to return the appropriate response, whose parameters are shown on table 4.4.
- **runQuestion()** in this function, we will simply run the question once through ChatScript, and then start the main command processing.
- **runCommands()** Given the NL response from ChatScript, it will split the Out of Band commands and start processing each one of them, as well as their responses, adding commands to the queue as needed.
- **processSolr()** For the Solr command, this method will construct the solr query, and keep sending it to Solr increasing its fuzziness level, until an answer is found, or a maximum fuzziness is reached, in which case it will return the empty Solr response.
- **splitCommands()** Taking a NL sentence as a parameter, this method will return both a list with all the Out of Band commands in the sentence, as well as the NL part of the sentence, or an empty string if the sentence was made only of Out of Band commands.
- **processGambit()** when a direct search does not return any answer, the system will perform a broader search in Solr. This is further explained in section 4.7.
- **sendChatScript()** This function will process the interaction with ChatScript, both sending the questions and handling the responses. For more details, see section 4.6.
- **sendSolrDefault()** Given a question and no other parameters or information, this function will send said question to Solr, so it will go through the default processing. This is mainly unused.
- **sendSolrEDisMax()** When a gambit is needed, send the question to Solr using an eDisMax query, which will search in different fields, valuing each field using a given weight.
- **sendSolr()** Used by the rest of the Solr related methods, this function will handle sending the payload given as parameter to Solr, and returning the JSON response.

Parameter	Name	Description
question	User's question	The question submitted by the user.
bot	Bot	The bot the query will be send to.
username	The user	A random identifier for the user communicating with the bot.

Table 4.3: Parameters in the query received by the front end controller.

Parameter	Name	Description
Answer	The Bot's response	The NL response generated by the bot.
Resource	An URL	The URL of the relevant document where the information is, if existent.

Table 4.4: Parameters in the query sent back to the client.

Finally, we will describe the Out of Band commands that the system will be able to process.

- **sendSolr** this command is issued whenever a question matches the pattern specified to request information about a Java topic.
- **solrResponse** after a search is performed in Solr, this command is issued to indicate whether or not a response has been found, and returning said response.
- **solrLinks** a special Solr search, looking for related topics in Solr.
- **solrLinksResponse** as a response to the solrLinks command, returns a list with the related topics.
- **gambit** perform an eDisMax search in Solr, using the full user question.
- **gambitResponse** returns the title of the most relevant document found in the eDisMax search.
- **gambitUnknown** issued when no document with a high enough score is found in Solr.

- **resource** sets the URL to be displayed in the final response returned to the client.
- **label** sets the title of the found document as the label for the response.

The syntax of the commands is specified in table 4.5.

Command	**Syntax**	**Description**
sendSolr	sendSolr *reqfield doctitle*	Searchs in solr for the *doctitle* and returns the *reqfield* field.
solrResponse	solrResponse *unknown*	The requested document was not found in Solr.
	solrResponse *reqfield response*	Returns as *response* the data of the field *reqfield*.
solrLinks	solrLinks *linklist*	Asks for a search in Solr for the name of the topics given as an uri in the linklist
solrLinksResponse	solrLinksResponse *nameslist*	Sets the response for the solr-Links command, returning the first name of the links.
gambit	gambit *topic*	Asks for a eDisMax search on Solr, passing the full question.
gambitResponse	gambitResponse *gambittopic*	After performing an eDisMax search, returns *gambittopic* as the suggested topic.
gambitUnknown	gambitUnknown	After performing an eDisMax search, indicates that no relevant document has been found.
resource	resource *URL*	Sets *URL* as the resource to be displayed in the client
label	label *topic*	Sets *topic* as the concept of the response

Table 4.5: Parameters in the query sent back to the client.

4.6 Chatbot

The Chatbot handles the processing of the natural language input from the user, and controls the conversation. To do so, it uses the chat engine ChatScript, described in section 2.2.2. We will first describe the Chatbot rules, and how they process the user input, and then proceed to describe how to launch and communicate with the ChatScript server.

4.6.1 The rules

Chatscript responds to a series of rules, specified in its topic files. These rules will match the user input and produce an appropriate response, or output a rejoinder if no rule is matched. A more in depth description of how Chatscript rules work can be found in section 2.2.2.

We have separated our topics across several files, each containing related topics, as well as the control script for the bot. We will start describing the control process, shown in listing 4.5

```
topic: ~control system ()
# on startup, do introduction
u: ( %input<%userfirstline)
    gambit(~tsaludos)

u: (< shut up >) $shutup = 1
u: (< talk  >) $shutup = null

u: () # main per-sentence processing

    $$currenttopic = %topic     # get the current topic at start of volley

    if ( %response == 0 ) {nofail(TOPIC ^rejoinder())}  # try for
        rejoinders. might generate an answer directly from what we are
        looking for.

    # Check if it is a java question
    if ( %response == 0) {nofail(TOPIC ^respond(~JAVA))}
    if ( %response == 0) {nofail(TOPIC ^respond(~EJEMPLOS))}


    if (%length == 0 AND %response == 0 )
    {
```

```
    nofail(TOPIC ^gambit($$currenttopic))  # gambit current topic since
         no input (usually start of conversation)
}

if (%response == 0) { nofail(TOPIC ^respond($$currenttopic)) } #
    current topic tries to respond to his input

if (%response == 0) # see if some other topic has keywords matching his
     input (given we have no response yet)
{
    @8 = ^keywordtopics()   # get topics referred in input
    loop()
    {
        $$topic = first(@8subject)
        nofail(TOPIC ^respond($$topic))
        if (%response != 0) # stop when we find something to say
        {
            ^end(RULE)  # we are done, this terminates the loop (not
                the rule)
        }
    }
}

# if we have rejoinders for what we said OR we asked a question, stop
    here
if (%outputrejoinder)
{
    end(TOPIC)
}

if (%response == 0 AND ^marked($$currenttopic)) { nofail(TOPIC ^gambit(
    $$currenttopic)) } # gambit current topic since keywords match
    current topic

# if no topic reacts, go to the TSALUDOS keyworldless topic
if (%response == 0)
{
    nofail(TOPIC ^respond(~TSALUDOS))
}

if (%response == 0){ nofail(TOPIC ^gambit($$currenttopic)) } # gambit
    from current topic even though no  keywords matched


if (%response == 0)
```

```
    {
         ^repeat()
      [Lo siento, no te he entendido. Podrias reformularlo, por favor?]
     [Perdona, no te he entendido bien. Decias?]
     [Eins? Podrias repetir eso ultimo?]
  }
```

Listing 4.5: Control process for ChatScript

In this process, we first check if we are currently in any conversation and have any pending rejoinders ready to match the output. Then proceed to check for rules looking for matches in the JAVA and EJEMPLOS topics. If no answer is provided, we will look for gambits and proper responses in the current topic, and then in the rest of the topics with keywords associated. In the case no match have been found yet, the user input will be test against the keywordless topic, looking for both responses and gambits, and, finally, if there is still no response, a generic answer will be provided, asking the user to modify its question.

As we have just mentioned, there are several topics, both with and without keyword, spread across several files, containing both said topics and the sets of concepts used in the bot. These files and the topics contained in them are:

- **topic.top** This file has most of the concepts used in the other files, as well as three generic topics:
  - *TENCUESTA*[5] to answer questions regarding the poll that will be presented to the user.
  - *EXAMENES* containing generic responses to questions about the exams.
  - *INTRO* with information that will allow the bot to identify himself.
- **java.top** This file contains the two main topics for answering the Java questions.
  - *JAVA* will answer the questions related to Java concepts, as well as produce gambits and question the user when the bot has the control over the conversation.
  - *EJEMPLOS* a slight variation from the previous topic, this will handle example requests from the user.

[5]Since we had a concept defined as "encuesta", we needed to differentiate this topic, thus the starting "T"

Field	Description
user	The string identifying the user performing the question.
bot	The bot that the question is directed to, in case there are several bot available in the server.
question	The user question

Table 4.6: Fields for the request to ChatScript

- **javaconcepts.top** contains the list of all the Java concepts that the bots knows about. It is automatically generated from the data stored in Solr.
- **insults.top** with the topic of the same name (INSULTOS), responds to insults and bad words input by the user.
- **estado.top** a single topic file, responsible of responding when the user enters information about himself, or asks the bot about its status.

All these files need to be compiled into binary data to be used by the ChatScript server.

4.6.2 The server

ChatScript provides two ways of interaction. For debugging purposes, it has a command line interface, and can also be deployed as a service listening in a TCP socket for user input. This server is what we have choose to use to interact with ChatScript. The full deployment instructions can be found in the appendix, and we will proceed to describe here the communication process.

As stated ChatScript listens on a TCP socket, waiting for requests containing three null separated strings, described in table 4.6

The server will then return in the same connection the response generated using the process previously explained, and close the connection. If a new interaction is needed, a new socket will be opened, following the same process.

4.7 Solr instance

In this section we will describe how the Solr instance is set up, as well as the schema and the search procedure. We are using Apache Solr 4.10.2 as a document and search engine. The search queries will be send by the controller described in section 4.5. The server will contain the data recovered from the Vademecum, structured in documents (one for each topic), and will allow us to perform the required searches. We will first describe the schema for the aforementioned documents,

4.7.1 Data schema

The scrapped data is stored using a Solr core containing every relevant document. This is done by describing the structure of said documents in the schema.xml for Solr. In this file, we can consider three main groups of fields:

- **Stock solr fields:** These fields are internal for Solr, and we will not describe them here.
- **Document fields:** the fields scrapped from the Vademecum, they are described in table 4.7.
- **titleDefinition field:** this field is generated concatenating the definition and title fields, to facilitate general search.

The XML for these fields is as in listing 4.6

```
 <!-- Fields for elearning -->

<field name="title" type="text_search_es" indexed="true" stored="true"
    multiValued="false"/>
<field name="alternative" type="lowercase" indexed="true" stored="true"
    multiValued="false"/>
<field name="concept" type="lowercase" indexed="true" stored="true"
    multiValued="false"/>
<field name="resource" type="string" indexed="true" stored="true"
    multiValued="false"/>

<!-- First sentence of the scrapped text -->
<field name="definition" type="text_search_es" indexed="true" stored="
    true" multiValued="false"/>
```

```

<!-- Full text -->
<field name="description" type="text_search_es" indexed="true" stored="
    true" multiValued="false"/>

<!--relations -->
<field name="links" type="text_ws" indexed="true" stored="true"
    multiValued="true"/>

<!-- Each topic can have different examples -->
<field name="examples" type="text_general" indexed="false" stored="true"
    multiValued="true"/>

<!-- Field for text search -->
<field name="titleDefinition" type="text_es" indexed="true" stored="false
    " multiValued="true" termVectors="true" termPositions="true"
    termOffsets="true" />
<copyField source="title" dest="titleDefinition"/>
<copyField source="definition" dest="titleDefinition"/>
```

Listing 4.6: fields defined for the Java documents in the schema

As shown in the table, the fields have a "field type" associated, that will determine how they are stored and how they are tokenized and processed when performing the indexing and the search. The fieldTypes we are using are as follow:

- **string** This field is a default Solr field, and stored verbatim using the solr.StrField class.
- **text_general** A default Solr field, using the sorl.TextField class.
- **lowercase** A variation on the default lowercase fieldType by Solr, it used the default keyword tokenizer and the lowercase filter, and we have added the ASCIIFolding filter.
- **text_search_es** Based on Solr's Spanish fields, this field will contain general text to perform a search. It uses a standard tokenizer, as well as the following filters:
  - Lowercase filter: converts all words to lowercase
  - Stopfilter factory using Spanish stopwords and the snoball stemming algorithm
  - Spanish light stem filter, a default solr stemmer for Spanish.

Field	**Field Type**	**Description**
title	text_search_es	The title of this particular document.
alternative	lowercase	If exists, the English name for the document.
concept	lowercase	What concept does this document refers to.
resource	string	The link for this document.
definition	text_search_es	The first sentence of the document.
description	text_search_es	The text of the document.
links	text_ws	Related documents to this one.
examples	text_general	The scrapped text of the examples.

Table 4.7: Fields for the documents stored in the Solr schema.

– Only for the query processing, a worddelimiter filter to do the QA.

The listing 4.7 shows the description of the text_search_es fieldType.

```xml
<!-- Based on Solr default spanish fields-->
<fieldType name="text_search_es" class="solr.TextField"
    positionIncrementGap="50">
   <analyzer type="index">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
            words="lang/stopwords_es.txt" format="snowball" />
      <filter class="solr.SpanishLightStemFilterFactory"/>
   </analyzer>
   <analyzer type="query">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <!-- Lowecase and spanish stemming-->
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
            words="lang/stopwords_es.txt" format="snowball" />
      <filter class="solr.SpanishLightStemFilterFactory"/>
      <!--<filter class="solr.EdgeNGramFilterFactory" minGramSize="2"
          maxGramSize="15" side="front"/>-->
      <!-- Replace puntuaction marks -->
      <filter class="solr.WordDelimiterFilterFactory"
            generateWordParts="1"
            splitOnCaseChange="0"
            splitOnNumerics="0"
            stemEnglishPossessive="0"
            />
   </analyzer>
</fieldType>
```

Listing 4.7: Definition for the text_search_es fieldType

With this configuration we will be able to do the queries described in the following sections.

4.7.2 Faceted query

In the event that ChatScript identifies the question and the topic the user is asking for, we will perform a faceted search in Solr, looking for the fields requested in the Out of Band command. The query will be done in JSON format. An example of a query is shown in listing 4.8, and the meaning of each field is described next:

```
{
  "q" : "title:for~0",
  "wt" : "json",
  "fl" : "*,score",
  "rows" : "1"
}
```

Listing 4.8: Example JSON query for Solr

- **q** contains the actual query sent to the server, with the filter for the query and the fuzziness. In the example, we are searching for documents containing the given string in the title, with a fuzziness of 0 (an exact match).
- **wt** the format the data will be returned in. In the example, we want the data in JSON format.
- **fl** the list of fields we want for the documents in the response. In the example, this is set to all the fields in the document, specified in the query as a wild card "*", as well as the score for the match.
- **rows** the number of documents to return.

With this search, we will try to look up the documents when the ChatScript module clearly identifies the topic the user is asking about, and, properly defining the returning fields for the search, the controller will show the relevant data. In case the question is not clearly identified, but ChatScript recognizes the user is talking about some topic related to Java, this query won't be valid, and we will have to perform an eDisMax query, described in the next section.

4.7.3 Gambit query

When ChatScript identifies a sentence talking about a Java concept, but does not recognize a question, we will perform a search in Solr using the entire question, treating it as a QA system. To do so, we will perform an eDisMax query, taking advantage of the stemmers and tokenizers we have configured in section 4.7.1. This type of query is designed to process user input directly, searching for the keyword across multiple fields, with different boosts based on the significance of each field, and allowing multiple options to influence the score on a case to case basis.

Like the regular Solr query, this query is perform using a JSON format, and sent to the Solr service as the payload to a GET request. An example of an eDisMax query is shown in listing 4.9

```
{
  "q": "hablame de los bucles for",
  "defType": "edismax",
  "qf": "title^10.0 description^2.0",
  "fl": "*,score",
  "rows": "1",
  "wt": "json",
  "lowercaseOperators": "true",
  "stopwords": "true"
}
```

Listing 4.9: Example eDisMax query for Solr

The significance of each in the query is as follows:

- **q** The question as sent by the user, with no processing.
- **defType** Explicitly set the query type as eDisMax.
- **qf** Set the weights for each field to consider in the query.
- **fl** The fields to return.
- **rows** The number of documents to return.
- **wt** The format for the response.
- **lowercaseOperators** Interpret lowercase words as boolean operators, such as "and" and "or".
- **stopwords** Use the stopwords defined in the schema.

This search will provide a broader match than the faceted query, finding matches when the topic of the question is not clear, and offering the answer to the user. To prevent completely unrelated topics to be offered to the user, the score is retrieved and only the answers with a minimum score will be returned.

4.8 Summary

In this Chapter, we have described a prototype of the proposed architecture for a system focussing on supporting lessons about the Java Programming Language.

We started providing a general overview of the system, describing how each module is related to the others.

We then talked about the *information retrieval* module, and how the information about the Java programming language was recovered from the Java Vademecum, using scrapy, and how we mapped it to a RDF taxonomy. We then studied the JSON format that would be used to upload the data to Solr.

We moved onto the *web interface* that presents the system to the final user, explaining the interface and how the system works to the final user. We also presented the interactions of the client with the server.

The client connects with the *Front end controller*, and we studied its implementation. We showed how the controller connects with the other modules, as well as analysed the Out of Band commands the different modules use to communicate with each other, and the workflow of the controller.

Taking a look at the *conversational agent*, we studied ChatScript and the rules used in our system, as well as how they were triggered and the output they produce.

Finally, we analysed the *Solr instance* configuration, taking a close look at the schema and the fields for the stored documents. We then proceeded to discuss the types of queries the system, the regular faceted query for user questions, and the eDisMax queries for gambits.

CHAPTER 5

Case study: GSI Bot

In this chapter, we will describe the prototype of a bot we developed for the GSI web page following the architecture described in chapter 3. We will first present the process to recover the data as Linked Data, to them describe the interface and the modules of the system.

5.1 Overview of the system

Along with the prototype described in chapter 4, we have also developed a system with the data from the GSI web page, including data from the projects, publications and staff. For this, we have similar modules:

1. A Javascript client acting as the user interface.
2. A Python controller, handling the flow of the information in the system
3. A different ChatScript bot, handling the conversation.
4. An Apache Solr core, with all the data.

In this prototype, the data was recovered using a mix of techniques, and integrated into a single core in Solr.

5.2 Recovering and storing the data

Similarly to the previous chapter, the data for this prototype has been recovered and converted into RDF and JSON formats,

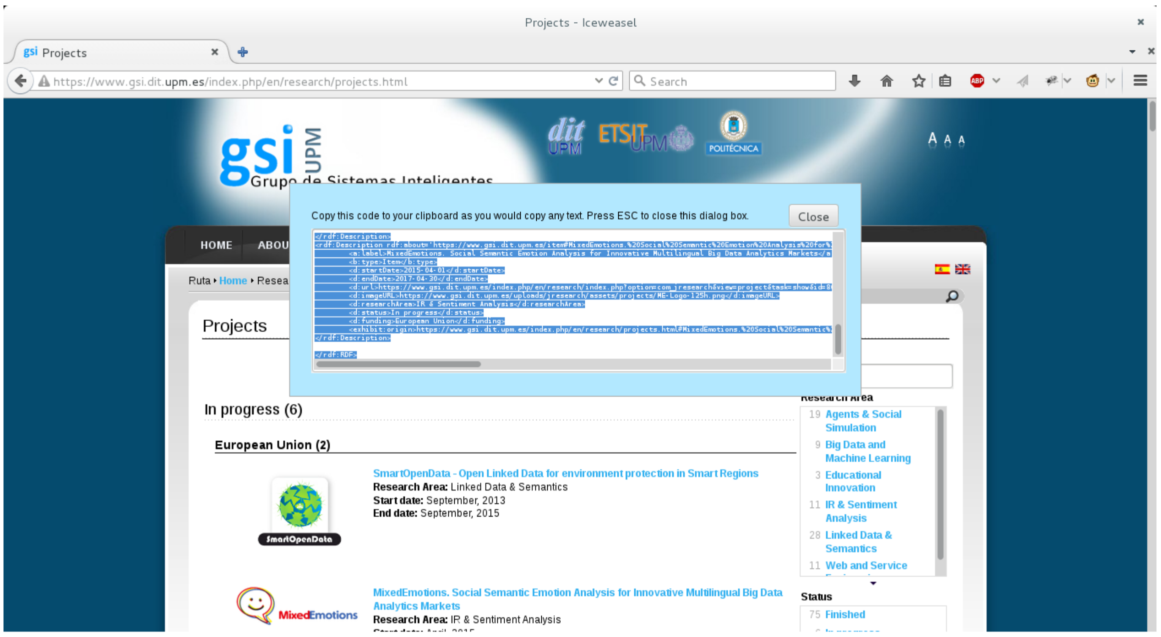
We considered three types of data from the GSI web page: the information about the members of the group, their publications and the projects the group has taken part of. Each type comes from a different part of the website, and therefore will be considered independently.

5.2.1 Projects

For the projects information, the data is available in different formats in the web page itself, including RDF/XML, as shown in figure 5.1

5.2.2 Publications

Each publication listed in the GSI web page has an attached bibtex citation. Therefore, it is possible to use a bibtex ontology[1] to map the elements in the bibtex files to semantic data. The mapping for the classes is shown in table 5.1.


Copy this code to your clipboard as you would copy any text. Press ESC to close this dialog box.

```
</rdf:Description>
<rdf:Description rdf:about='https://www.gsi.dit.upm.es/item#MixedEmotions.%20Social%20Semantic%20Emotion%20Analysis%20for%
        <a:label>MixedEmotions. Social Semantic Emotion Analysis for Innovative Multilingual Big Data Analytics Markets</a
        <b:type>Item</b:type>
        <d:startDate>2015-04-01</d:startDate>
        <d:endDate>2017-04-30</d:endDate>
        <d:url>https://www.gsi.dit.upm.es/index.php/en/research/index.php?option=com_jresearch&view=project&task=show&id=8
        <d:imageURL>https://www.gsi.dit.upm.es/uploads/jresearch/assets/projects/ME-Logo-125h.png</d:imageURL>
        <d:researchArea>IR & Sentiment Analysis</d:researchArea>
        <d:status>In progress</d:status>
        <d:funding>European Union</d:funding>
        <exhibit:origin>https://www.gsi.dit.upm.es/index.php/en/research/projects.html#MixedEmotions.%20Social%20Semantic%
</rdf:Description>

</rdf:RDF>
```

Projects - Iceweasel
https://www.gsi.dit.upm.es/index.php/en/research/projects.html
Close
HOME
Projects
In progress (6)
European Union (2)
SmartOpenData - Open Linked Data for environment protection in Smart Regions
Research Area: Linked Data & Semantics
Start date: September, 2013
End date: September, 2015
MixedEmotions. Social Semantic Emotion Analysis for Innovative Multilingual Big Data Analytics Markets
Research Area: IR & Sentiment Analysis
19 Agents & Social Simulation
9 Big Data and Machine Learning
3 Educational Innovation
11 IR & Sentiment Analysis
28 Linked Data & Semantics
11 Web and Service
Status
75 Finished

Figure 5.1: RDF exporter for the projects.

Bibtex tag	Ontology mapping	Description
article	bibtex:Article	An article from a journal or magazine.
book	bibtex:Book	A book with an explicit publisher.
conference	bibtex:Conference	An article in a conference proceedings.
inbook	bibtex:Inbook	A part of a book, which may be a chapter (or section or whatever) and/or a range of pages.
incollection	bibtex:Incollection	A part of a book having its own title.
masterthesis	bibtex:Masterthesis	A Master's thesis
phdthesis	bibtex:Phdthesis	A PhD thesis.
proceedings	bibtex:Proceedings	The proceedings of a conference
techreport	bibtex:Techreport	A report published by a school or other institution, usually numbered within a series.

Table 5.1: Classes for the bibtex documents.

As seen in listing 5.1, the mapping for the properties follows a simple pattern, similar to the mapping of the classes. To this mapping we have added the source of the document using Dublin Core, so the original bibtex can be referenced as needed.

```
<bibtex:Conference rdf:about="gsi:serranoTwitter2015">
   <bibtex:hasTitle>A survey of Twitter Rumor Spreading Simulations</
      bibtex:hasTitle>
   <bibtex:hasAuthor>gsi:eserrano</bibtex:hasAuthor>
   <bibtex:hasAuthor>gsi:ciglesias</bibtex:hasAuthor>
   <bibtex:hasAuthor>gsi:mgarijo</bibtex:hasAuthor>
   <dcterms:source>http://www.gsi.dit.upm.es/index.php/es/investigacion/
      publicaciones.bibtex?controller=publications&amp;task=export&amp;
      id=364</dcterms:source>
   <bibtex:hasYear>2015</bibtex:hasYear>
   <bibtex:hasMonth>September</bibtex:hasMonth>
   <bibtex:hasBooktitle>7th International Conference on Computational
      Collective Intelligence Technologies and Applications</bibtex:
      hasBooktitle>
 </bibtex:Conference>
```

Listing 5.1: Example bibtex document converted to RDF

5.2.3 People

Finally, the information available in the GSI web page about the members of the group cannot be exported in any format, so we have used scrappy, described in section 2.5.1 to crawl the data and export it into an appropriate format. In order to do so, we used the Friend of a Friend (FOAF)[2] ontology as well as some elements from the Semantically-Interlinked Online Communities (SIOC) ontology[3] to represent each person in the group. An example mapping can be found in listing 5.2

```
<foaf:Person rdf:about="gsi:amardomingo">
  <foaf:workInfoHomepage>http://www.gsi.dit.upm.es/index.php?option=
     com_jresearch&amp;view=member&amp;task=show&amp;id=84</foaf:
     workInfoHomepage>
```

[1]http://purl.oclc.org/NET/nknouf/ns/bibtex
[2]http://xmlns.com/foaf/spec/
[3]http://rdfs.org/sioc/spec/

```
  <sioc:Role>5&gt; Becario de Grado</sioc:Role>
  <foaf:givenName>Alberto Mardomingo Mardomingo</foaf:givenName>
  <foaf:homepage>http://gsi.dit.upm.es/~amardomingo/</foaf:homepage>
  <foaf:img>http://www.gsi.dit.upm.es/uploads/jresearch/assets/members/
      Foto.jpg</foaf:img>
</foaf:Person>
```

Listing 5.2: Example semantic data about a member of the group

This data was generated using scrappy, running the extractor shown in listing 5.3.

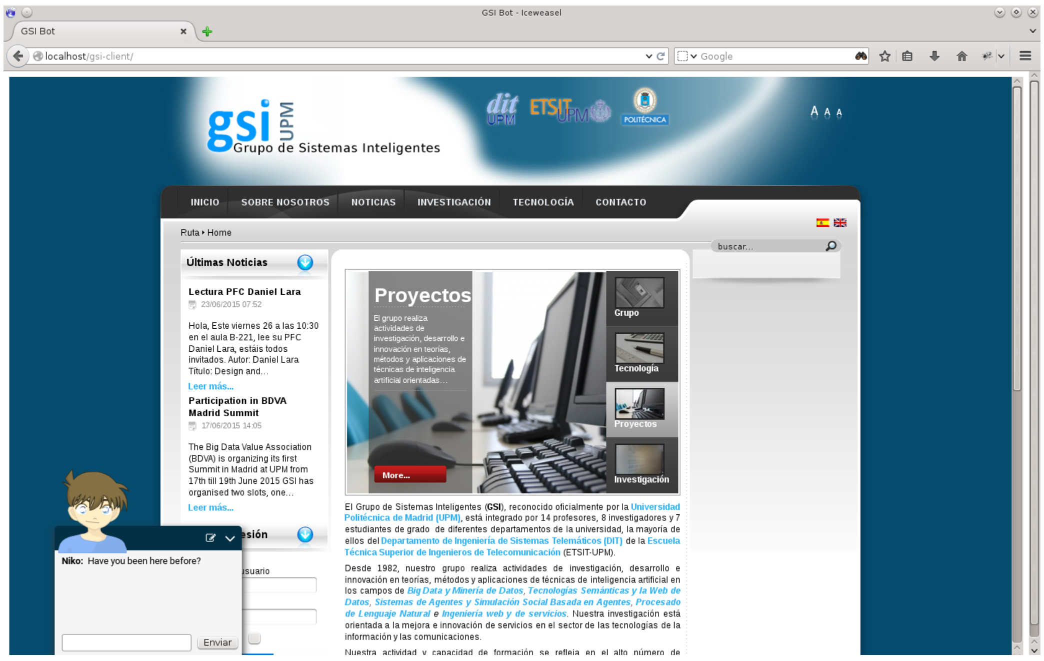
```
_:gsipeople:
  rdf:type: sc:Fragment
  sc:type: foaf:Person
  sc:selector:
    *:
      rdf:type: sc:UriPatternSelector
      rdf:value: "http://www.gsi.dit.upm.es/index.php?option=com_jresearch&
          view=staff&layout=positions"
  sc:identifier:
    *:
      rdf:type: sc:BaseUriSelector
  sc:subfragment:
    *:
      sc:type: foaf:Person
      sc:selector:
        *:
          rdf:type: sc:CssSelector
          rdf:value: ".jrperson"
      sc:identifier:
        *:
          rdf:type: sc:CssSelector
          rdf:value: "a"
          sc:attribute: "href"
      sc:subfragment:
        *:
          sc:type: rdf:Literal
          sc:relation: foaf:givenName
          sc:selector:
            *:
              rdf:type: sc:CssSelector
              rdf:value: "a"
_:gsiperson:
```

```
rdf:type: sc:Fragment
sc:type: foaf:Person
sc:selector:
  *:
    rdf:type: sc:UriPatternSelector
    rdf:value: "http://www.gsi.dit.upm.es/index.php?option=com_jresearch&
        view=member&task=show&id=*"
sc:identifier:
  *:
    rdf:type: sc:BaseUriSelector
sc:subfragment:
  *:
    sc:type: rdf:Literal
    sc:relation: foaf:homepage
    sc:selector:
      *:
        rdf:type: sc:CssSelector
        rdf:value: ".personalpage a"
  *:
    sc:type: rdf:Literal
    sc:relation: foaf:img
    sc:selector:
      *:
        rdf:type: sc:CssSelector
        rdf:value: ".persimg"
        sc:attribute: "href"
  *:
    sc:type: rdf:Literal
    sc:relation: foaf:phone
    sc:selector:
      *:
        rdf:type: sc:CssSelector
        rdf:value: ".jrpf"
   *:
    sc:type: rdf:Literal
    sc:relation: sioc:Role
    sc:selector:
      *:
        rdf:type: sc:CssSelector
        rdf:value: ".jrposition"
   *:
    sc:type: rdf:Literal
    sc:relation: foaf:based_near
    sc:selector:
      *:
        rdf:type: sc:CssSelector
        rdf:value: ".jrlocation"
```

Listing 5.3: Extractor for the GSI people section

5.3 User interface

For this system, the user interface is similar to the one described in section 4.4, with some minor changes.


GSI Bot - Iceweasel

GSI Bot

localhost/gsi-client/

Grupo de Sistemas Inteligentes

INICIO SOBRE NOSOTROS NOTICIAS INVESTIGACIÓN TECNOLOGÍA CONTACTO

Ruta ‣ Home

Últimas Noticias

Lectura PFC Daniel Lara
23/06/2015 07:52

Hola, Este viernes 26 a las 10:30 en el aula B-221, lee su PFC Daniel Lara, estáis todos invitados. Autor: Daniel Lara Título: Design and…

Leer más...

Participation in BDVA Madrid Summit
17/06/2015 14:05

The Big Data Value Association (BDVA) is organizing its first Summit in Madrid at UPM from 17th till 19th June 2015 GSI has organised two slots, one…

Leer más...

Proyectos

El grupo realiza actividades de investigación, desarrollo e innovación en teorías, métodos y aplicaciones de técnicas de inteligencia artificial orientadas…

More...

Grupo
Tecnología
Proyectos
Investigación

El Grupo de Sistemas Inteligentes (**GSI**), reconocido oficialmente por la Universidad Politécnica de Madrid (UPM), está integrado por 14 profesores, 8 investigadores y 7 estudiantes de grado de diferentes departamentos de la universidad, la mayoría de ellos del Departamento de Ingeniería de Sistemas Telemáticos (DIT) de la Escuela Técnica Superior de Ingenieros de Telecomunicación (ETSIT-UPM).

Desde 1982, nuestro grupo realiza actividades de investigación, desarrollo e innovación en teorías, métodos y aplicaciones de técnicas de inteligencia artificial en los campos de *Big Data y Minería de Datos*, *Tecnologías Semánticas y la Web de Datos*, *Sistemas de Agentes y Simulación Social Basada en Agentes*, *Procesado de Lenguaje Natural* e *Ingeniería web y de servicios*. Nuestra investigación está orientada a la mejora e innovación de servicios en el sector de las tecnologías de la información y las comunicaciones.

Nuestra actividad y capacidad de formación se refleja en el alto número de

Niko: Have you been here before?

usuario

Enviar

Figure 5.2: Web interface for the client.

5.4 Controller

The controller handles the requests received from the client described in the previous section, interacting with the required modules as needed, and interpreting the Out of Band commands generated by each module. This module is very similar to the one described in section 4.5 of the previous chapter, and is developed using the same technologies. It follows the same functional model described in section 4.5.1.

5.4.1 Structural Model

In this section we will describe the structure followed by the controller of this prototype. The relevant methods of said structure are shown in figure 5.3.

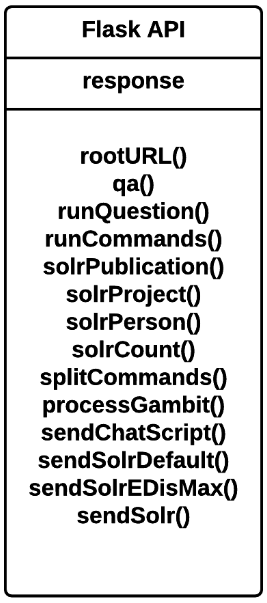


Figure 5.3: Front end controller structure

The function for each of those methods is described next:

- **rootURL()** triggered when the base URL for the controller is requested, it acts in the same way as the qa() method.
- **qa()** Parsing the request parameters shown in table 4.3, this method will start the process to process the user question, and return the response once said process is complete, following the structure on table on table 4.4.
- **runQuestion()** Send the question to ChatScript once, to generate the first Out of Band commands and start processing them.
- **runCommands()** Parse the response from ChatScript and split the Out of Band commands, in order to start processing each one of them, as well as their responses, adding commands to the queue as needed.

- **solrPublication()** When the user is asking about a publication, parse the parameters and perform the relevant Solr lookup.
- **solrProject()** Parse handle the queries to Solr when the question involves projects.
- **solrPerson()** Handle requests regarding members of the GSI.
- **solrCount()** When the user ask about quantities rather than about the documents themselves, perform the appropriate Solr lookup.
- **processGambit()** As a last resource, perform a broad search in Solr with the user query. This is further explained in section 4.7.
- **sendChatScript()** This function will process the interaction with ChatScript, both sending the questions and handling the responses. For more details, see section 4.6.
- **sendSolrDefault()** Given a question and no other parameters or information, this function will send said question to Solr, so it will go through the default processing. This is mainly unused.
- **sendSolrEDisMax()** Send the question to Solr using an eDisMax query, usually for a gambit.
- **sendSolr()** Send a direct query to Solr. Mainly used by the rest of the Solr lookup methods.

Finally, we will describe the Out of Band commands that the system will be able to process.

- **solrPublication** and **solrResponsePublication** handle the interactions when the user question is about the publications.
- **solrProject** and **solrResponseProjects** are issued for the query and response when performing lookups about the projects.
- **solrPerson** and **solrResponsePerson** handle the queries for the lookups related with the GSI members.
- **solrCount** and **solrcounted** perform the query for the number of documents relevant to a given filter.
- **solrLinks** a special Solr search, looking for related topics in Solr.
- **gambit** performs an eDisMax search in Solr, with the user query.

- **gambitResponse** returns the response of the gambit eDisMax query.
- **gambitUnknown** issued when the gambit does not return any relevant document.
- **resource** sets the URL to be shown to the user in the client.

5.5 Chatbot

Using the ChatScript chat engine, the Chatbot handles the conversation and process the natural language interactions with the user. However, to adapt to the fact that the documents stored in Solr for this prototype are not homogeneous, the rule structure has been changed.

5.5.1 The rules

We have separated the rules regarding different types of documents in one topic file per type, and therefore, we will have the following files and topics:

- **people.top** This file has the interactions regarding the members of the group.
- **projects.top** with the topics regarding the projects.
- **publications.top** contains the topics relevant to the publications.
- **introductions.top** contains simple chat about the bot, as well as greetings and farewells.
- **mixed.top** in this file are stored topics with questions regarding more than one of the fields.

Finally, the control script used for this bot is very similar to the one provided by default with ChatScript, since we do not have the limitations regarding changing the language in this bot.

This control script will first look for rejoinders in the current topic, to then proceed to offer a gambit if no rejoinder is found and the keyword match. If there is no gambit available for the current interaction, the system will look for standard responders, both in this topic and in any other topic whose keywords match the current volley. If no topic reacts, the systems will then try to match the responders from the keywordless topics, to finally gambit any keyword topic. If there is no response at that point, it forces a gambit from the current topic, to finally offer a generic response, indicating the system does not know how to respond.

5.5.2 The server

The ChatScript server for this prototype is deployed in the same way than the server in the previous chapter, explained in section 4.6.2. However, it is worth mentioning that a single ChatScript instance cannot hold both bots, so if the same physical (or virtual) server is to run both bots, it would be needed to run two ChatScript instances, each one of them in a different port.

5.6 Solr instance

In this section we will describe how the Solr instance has been configured for this system, as well as the search procedure. As in section 4.7, we are using Apache Solr 4.10.2, which will receive the queries from the controller described in section 5.4. The Solr instance will contain a core with the data scrapped from the web page, allowing us to have all the data in a single place.

The data was merged into a single RDF file, to then be imported and uploaded into the Solr instance.

5.6.1 Solr Schema

For this system, we have stored all the scrapped data in the same Solr core. Therefore, we can classify the fields according to the type of document they are associated with. We will consequently look at the fields grouping them according to said type of document:

- **Stock solr fields:** Fields internal for Solr, we will not describe them here.
- **Document fields:** the fields scrapped from the web.

  Fields for the projects: These fields are derived from the project exported structure.

  Fields for the members of the GSI: Describing the people at the GSI, these fields are based on FOAF fields.

  Fields for the publications: Based on the bibtex ontology described in section 5.2.2.

Field	Field Type	Description
about	lowercase	URI for the document being described
type	lowercase	RDF Class of the document
url	string	URL for associated filed for this document, like a personal page or the full text of a publication
searchField	text_search	A field with information from all the document types, to perform regular searches

Table 5.2: Common fields for different types of documents.

- **searchField field:** this field is generated concatenating fields from all three types of fields, to allow a default search with a single field.

A short description of each field and the associated field types can be seen in table 5.2 for the common general purpose fields, table 5.3 for the members of the group, table 5.5 for the projects and 5.4 for the publications.

The "text_search" field is based on the default English fields, and is defined as shown in listing 5.4

```
<fieldType name="text_search" class="solr.TextField"
    positionIncrementGap="50">
   <analyzer type="index">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
            words="lang/stopwords_en.txt" format="snowball" />
   </analyzer>
   <analyzer type="query">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
```

Field	**Field Type**	**Description**
givenname	text_search	The name of the person being described
homepage	string	The personal page for this person
img	string	A link to an image of this person
based_near	lowercase	The location for the person described
phone	string	Their phone number
workinfohome	string	The page in their workplace describing this person
role	text_search	The position for this person

Table 5.3: Fields associated with personal data

Field	**Field Type**	**Description**
Journal	lowercase	Journal for the described publication
volume	string	The volume this document appeared in
year	string	The year the document was published
month	lowercase	The month the document was published
title	text_search	The title for this document
note	text_search	A comment associated with the bibliographic citation
school	text_search	The name of the School where the document was written
series	string	The series for the publication this document appeared in
publisher	text_search	The publisher for the Journal of the document
number	string	The number of a work in a series
abstract	text_search	An abstract about the document described
address	lowercase	The address for the publisher or the school
editor	text_search	The editor for this document
author	text_search	The list of authors of this document
pages	string	The page numbers of the Journal in which this publication appeared
chapter	string	The chapter of a book for this publication
source	string	The bibtex original file

Table 5.4: Fields for the publications.

Field	Field Type	Description
label	text_search	The tittle for this project
status	lowercase	Whether the project is active or not
startdate	string	The date the project is supposed to start
enddate	string	The date project finished
funding	text_search	The origin of the funding for the project
imageurl	string	A link to an image associated with the project
researcharea	lowercase	The field this project is associated with
origin	lowercase	A document with all the info about this project

Table 5.5: Fields for the projects.

```
            <filter class="solr.StopFilterFactory" ignoreCase="true"
                    words="lang/stopwords_en.txt" format="snowball" />
            <filter class="solr.WordDelimiterFilterFactory"
                    generateWordParts="1"
                    splitOnCaseChange="0"
                    splitOnNumerics="0"
                    stemEnglishPossessive="0"
                    />
        </analyzer>
    </fieldType>
```

Listing 5.4: Definition for the text_search fieldType

With this configuration we will perform the queries described in the next sections.

5.6.2 Solr queries

For this system, we have considered several types of queries. We have considered questions about the different objects and its relations, questions about quantifying specific subsets, and gambit queries.

5.6.2.1 Questions about quantities

This query is performed when ChatScript identifies a question about the about of some type of document. This query will recover the number of documents in Solr matching the required criteria, and return it in an Out of Band command. For example, for a question about the number of publications in 2014, the query will be as shown in listing 5.5. The meaning of each field is described next:

```
{
  "q" : "type:*bibtex* AND year:2014",
  "wt" : "json",
  "rows" : "0"
}
```

Listing 5.5: Example JSON query for Solr

- **q** contains the actual query sent to the server, specifying the fields we are filtering with. In the example, we are searching for documents for year 2014. Since "year" is a field unique to publications, there is no real need to add the type filter, and is added only for clarity purposes.
- **wt** the format the data will be returned in. In the example, we want the data in JSON format.
- **rows** the number of documents to return. Since we only want the actual result count, rather than the documents, in the example it is set to 0.

This search will return a JSON with the number of documents matching the criteria, which in turn will be pass to ChatScript, generating the appropriate NL response.

5.6.2.2 General questions

ChatScript can also identify general questions about the different objects stored in Solr. In those cases, Solr will perform a regular search, returning the document with the highest score, which will them offered to the user as a response.

For example, if ChatScript identifies a question about the work the research group does about Linked Open Data, the query send to Solr will be as shown in 5.6. and will return the document with the highest score regarding the question, whether it is a publication or a project.

```
{
  "q" : "searchfield:linked\\ open\\ data",
  "wt" : "json",
  "rows" : "1"
}
```

Listing 5.6: Example query asking for Linked Open Data

A different search could be performed for people in the group. For example, if ChatScript receives a question asking for the publications of a user, two queries will be sent. The first one to identify the user, as shown in listing 5.7, which will return the "about" field with the identifier for the user, as well as the page with the list of publications for this user. The controller will then use the recovered identifier to search for publications that match the author, as shown in listing 5.8.

```
{
  "q" : "givenname:mardomingo",
  "wt" : "json",
  "fl": "about,workinfohomepage",
  "rows" : "1"
}
```

Listing 5.7: Query asking for the data about a user

```
{
  "q" : "author:gsi\:amardomingo",
  "wt" : "json",
  "fl": "about",
  "rows" : "10"
}
```

Listing 5.8: Query asking for the data about a user

5.6.2.3 Gambit queries

In the event that ChatScript is incapable of identifying the question the user is making, the system will perform an eDisMax query, looking for a match in the relevant fields for each document type, and offering the answer only if the score of the match is over a predetermined minimum score. The process is quite similar to the previous prototype, described in section 4.7.3.

5.7 Summary

In this chapter we have talked about a different prototype for our system, including documents with different structures, indexed into the system.

We first studied the three main sources of data, proposing a RDF mapping to export the scrapped data into semantic data. We also discussed the different techniques and tools used while recovering the data. We recovered the data from the GSI members, their publications and their projects.

Then we analysed the *user interface* for this prototype, very similar to the previous one.

We move on to analyse the new Front-end *controller*, which, although having the same functional model, has a different structural model, in order to account for the different structure of the documents indexed for this system. We analyse the new Out of Band commands, as well as their parameters.

Like with the controller, the new *conversational agent* is studied in this section, analysing the new file structure and control system.

To finish this chapter, we take a look at the Solr fields and schema built for this system, with the different analysers, accounting for the recovered documents being in English rather than Spanish. We also analyse the different queries that can be performed with this system.

CHAPTER 6

Evaluation

In this chapter we will analyse the behaviour and performance of the system. We will also evaluate the accuracy of its responses, and the end user experience compared to a regular QA system.

6.1 Overview

The systems developed for this project have multiple differentiate modules, each one of them with different hardware requirements. Therefore, we will first take a look at the performance for each module, to then focus in measuring the response of the entire system.

Together with the performance analysis, we have run a experiment to analyse the effect on the end user experience of the system, compared to a regular QA system, using the Java prototype [26]. To do so, we implemented a simple QA and then presented a reduced group of users with both systems, asking them to evaluate both systems.

6.2 Requirements and Benchmark

For our system, we will first analyse the memory and CPU usage under low load for each of the modules. The results can be seen in table 6.1.

The specifications of the system running the tests are shown in table 6.2

Figures 6.2 and 6.1 show CPU and memory usage for the controller and ChatScript over time. It clearly shows the difference between the system under low load, and the peaks in usage during the stress tests. Since Solr is deployed under Tomcat, it is not possible to plot its data independently from the rest of the Java processes, and we can only access the figures available in Solr's web interface, that indicate a consumption between 381.16MB and 788.50MB.

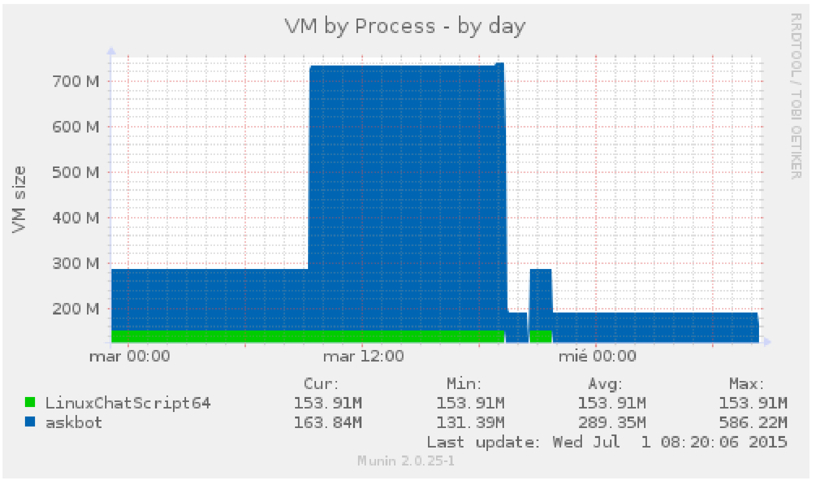
We have also measured the response time for the system for different queries. Figure 6.3 shows the measured times. We associate the three peaks shown in the graph, around 10ms, 40ms and 75ms with queries directly answered by ChatScript, such as social dialogue, queries with simple Solr lookups, and queries requiring a gambit (i.e. multiple Solr lookups)

Module	**CPU Usage**	**Memory**
ChatScript	0.1 %	152MB
Solr	1.9 %	533.74MB
Python Controller	0.4 %	153.91MB

Table 6.1: Memory and CPU usage for each system under low load

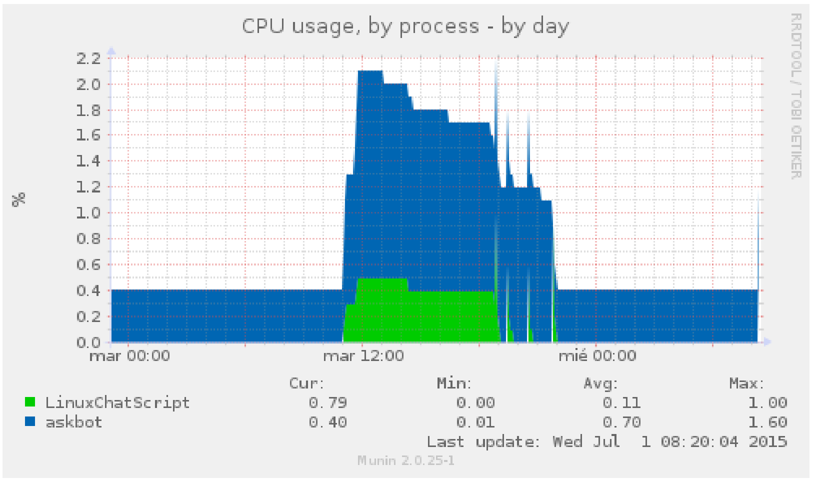
Operating System	Debian Jessie x64, Kernel 3.16.0-4
Motherboard	Asus V-P7H55E
CPU	Intel i5 650 3.20GHz
Memory	2x Kingston DDR3 1333MHz 4096MB
Hard Drive	Western Digital 7200rpm 1TB

Table 6.2: Memory and CPU usage for each system under low load



```
                        Cur:        Min:        Avg:        Max:
🟩 LinuxChatScript64    153.91M     153.91M     153.91M     153.91M
🟦 askbot               163.84M     131.39M     289.35M     586.22M
                        Last update:  Wed Jul  1 08:20:06 2015
```

Figure 6.1: Memory consumption from ChatScript and the Controller



```
                        Cur:        Min:        Avg:        Max:
🟩 LinuxChatScript      0.79        0.00        0.11        1.00
🟦 askbot               0.40        0.01        0.70        1.60
                        Last update:  Wed Jul  1 08:20:04 2015
```

Figure 6.2: CPU usage for ChatScript and the controller

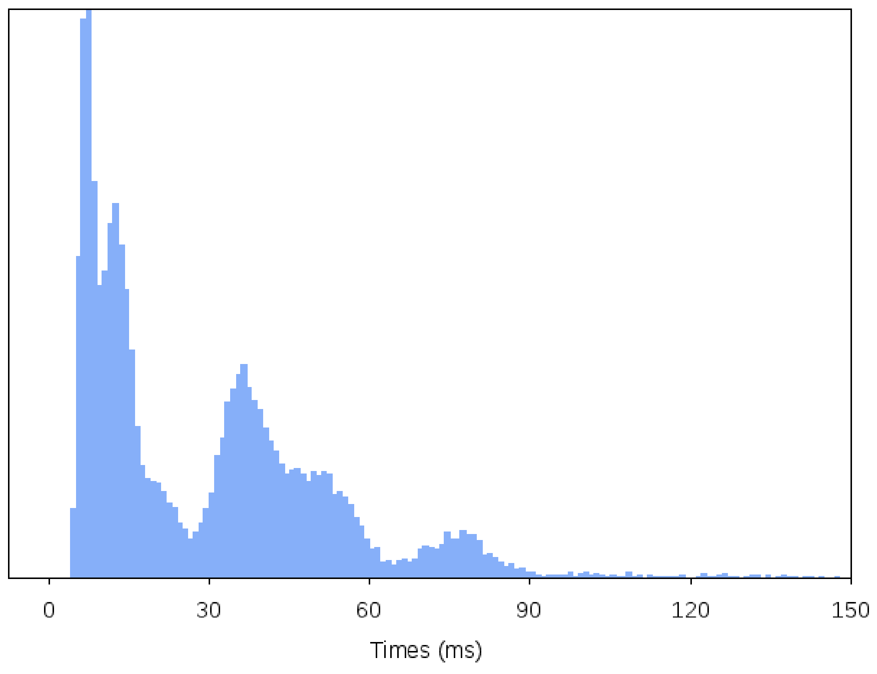


Figure 6.3: Times for queries

6.3 Corpus tests

During the development of the system, we prepared a corpus, to be able to test it in an automated way. The test corpus was made of questions and their expected responder, as well as the answer and resource for the expected topic. A short fragment of the corpus is shown in listing 6.1

```
Pregunta,Concepto,CS_RESPONSE,ANSWERER,RESOURCE
Hola,,"Hola",chatscript
que es un while?,while,sendSolr definition while,solr,http://www.dit.upm.es
    /~pepe/libros/vademecum/topics/302.html
que es un while,while,sendSolr definition while,solr,http://www.dit.upm.es
    /~pepe/libros/vademecum/topics/302.html
eres el profesor,,"bot",chatscript
```

Listing 6.1: Fragment of the test corpus built for the system.

The results of the tests using the corpus are shown in table 6.3

System	Success rate
Overall System	78.26%
Solr	80.43%
ChatScript	69.57 %

Table 6.3: Results for the test with the corpus

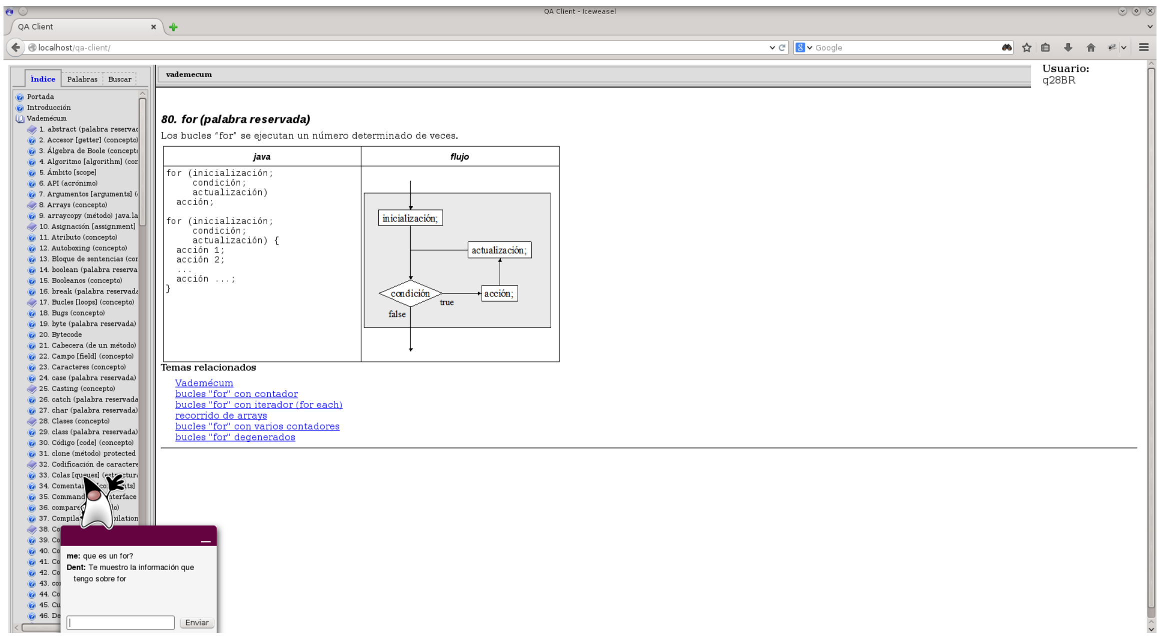
6.4 User experience

Finally, we tested the system in a trial with users, comparing the Java prototype to a regular QA system. The users are presented both interfaces, shown in figures 6.4 and 6.5, with a similar aspect to prevent the effect of different interfaces to have an impact on the results. The QA will search for the information in the indexed documents, and show the result in the screen, whereas our system will behave as described in chapter 4

The participants of the experiment ranged in age from 20 to 30 years old, most of them being under 25. 15% of them where female, and 69% of them where students. All of them had indicated they had experience with computers, and knowledge of the Java programming language, but none of them had participated in any previous study involving conversational agents.

6.4.1 Experiment procedure

We proposed the participants to interact with two different configurations of our prototype. They are presented a QA configuration and a Conversational Agent configuration (both with similar interfaces in order to avoid the effect of the possible confounding variable). The QA search for the information in the documents available and shows the result. With this configuration, the QA system is working as an IR since it does not extract any information from the document. This is simple but yet valid and useful configuration since the documents we worked with were relatively small. The second configuration consist on the conversational agent we developed in chapter 4, that have access to the same information as the former system (and uses the same IR module), with the added features of social dialogue and proactive recommendation of related topic and suggesting examples. In both cases, some answers will be served with suggestions of related topics and examples to ask for. Instead of implementing four different configurations of the system, this approach was


QA Client - Iceweasel
QA Client
localhost/qa-client/
Google
Índice
Palabras
Buscar
Portada
Introducción
Vademécum
1. abstract (palabra reservad
2. Accesor [getter] (concepto)
3. Álgebra de Boole (concepto
4. Algoritmo [algorithm] (cor
5. Ámbito [scope]
6. API (acrónimo)
7. Argumentos [arguments] (
8. Arrays (concepto)
9. arraycopy (método) java.la
10. Asignación [assignment]
11. Atributo (concepto)
12. Autoboxing (concepto)
13. Bloque de sentencias (cor
14. boolean (palabra reserva
15. Booleanos (concepto)
16. break (palabra reservada
17. Bucles [loops] (concepto)
18. Bugs (concepto)
19. byte (palabra reservada)
20. Bytecode
21. Cabecera (de un método)
22. Campo [field] (concepto)
23. Caracteres (concepto)
24. case (palabra reservada)
25. Casting (concepto)
26. catch (palabra reservada
27. char (palabra reservada)
28. Clases (concepto)
29. class (palabra reservada)
30. Código [code] (concepto)
31. clone (método) protected
32. Codificación de caractere
vademecum
inicialización;
actualización;
condición
acción;
true
false
Enviar

Usuario:
q28BR

80. for (palabra reservada)

Los bucles "for" se ejecutan un número determinado de veces.

java	*flujo*
`for (inicialización; condición; actualización) acción;` `for (inicialización; condición; actualización) { acción 1; acción 2; ... acción ...; }`	

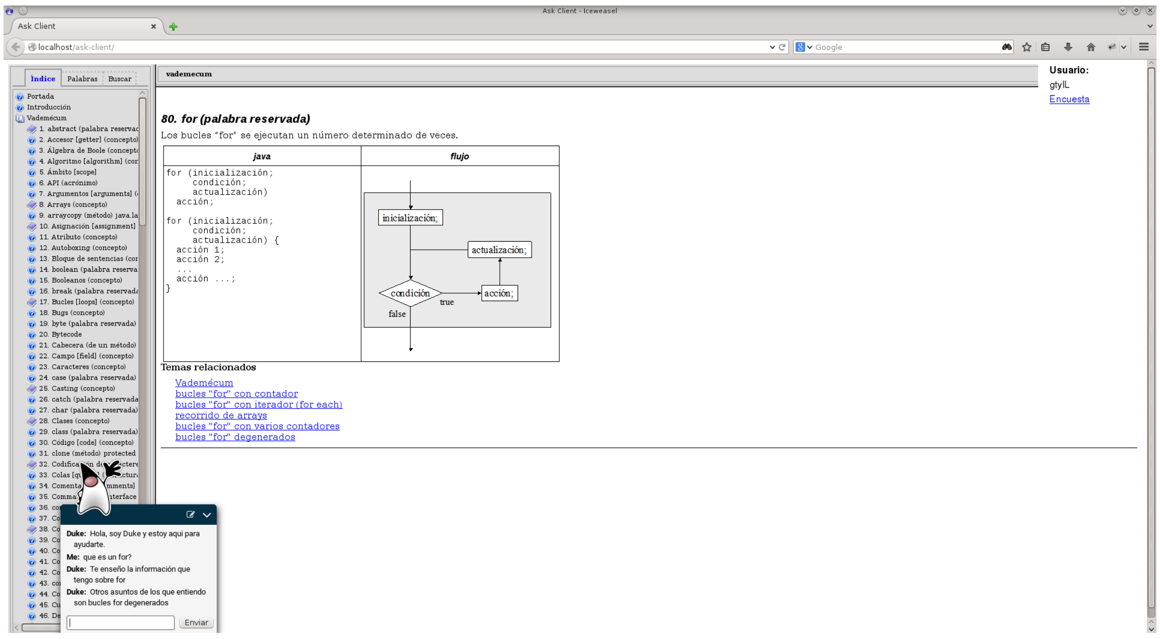
```
for (inicialización;
     condición;
     actualización)
  acción;

for (inicialización;
     condición;
     actualización) {
  acción 1;
  acción 2;
  ...
  acción ...;
}
```

Temas relacionados

Vademécum
bucles "for" con contador
bucles "for" con iterador (for each)
recorrido de arrays
bucles "for" con varios contadores
bucles "for" degenerados

me: que es un for?
Dent: Te muestro la información que tengo sobre for

Figure 6.4: QA interface after asking about the for loop


Ask Client - Iceweasel
Ask Client
localhost/ask-client/
Google
Índice
Palabras
Buscar
Portada
Introducción
Vademécum
1. abstract (palabra reservad
2. Accesor [getter] (concepto)
3. Álgebra de Boole (concepto
4. Algoritmo [algorithm] (cor
5. Ámbito [scope]
6. API (acrónimo)
7. Argumentos [arguments] (
8. Arrays (concepto)
9. arraycopy (método) java.la
10. Asignación [assignment]
11. Atributo (concepto)
12. Autoboxing (concepto)
13. Bloque de sentencias (cor
14. boolean (palabra reserva
15. Booleanos (concepto)
16. break (palabra reservada
17. Bucles [loops] (concepto)
18. Bugs (concepto)
19. byte (palabra reservada)
20. Bytecode
21. Cabecera (de un método)
22. Campo [field] (concepto)
23. Caracteres (concepto)
24. case (palabra reservada)
25. Casting (concepto)
26. catch (palabra reservada
27. char (palabra reservada)
28. Clases (concepto)
29. class (palabra reservada)
30. Código [code] (concepto)
31. clone (método) protected
vademecum
inicialización;
actualización;
condición
acción;
true
false
Enviar

Usuario:
gtylL
Encuesta

80. for (palabra reservada)

Los bucles "for" se ejecutan un número determinado de veces.

java / *flujo*

```
for (inicialización;
     condición;
     actualización)
  acción;

for (inicialización;
     condición;
     actualización) {
  acción 1;
  acción 2;
  ...
  acción ...;
}
```

Temas relacionados

Vademécum
bucles "for" con contador
bucles "for" con iterador (for each)
recorrido de arrays
bucles "for" con varios contadores
bucles "for" degenerados

Duke: Hola, soy Duke y estoy aqui para ayudarte.
Me: que es un for?
Duke: Te enseño la información que tengo sobre for
Duke: Otros asuntos de los que entiendo son bucles for degenerados

Figure 6.5: Interface for our system about the for loop

selected because the influence of the suggestions can still be measured, and otherwise the learning and fatigue effects of within-subjects evaluation with too many interfaces (that are very similar) may falsify the results.

We hypothesise that participants will prefer using natural language rather than keywords search to query the system, and that they will mostly follow the suggestions given, specially in the Conversational Agent configuration. Particularly we state the following hypothesis:

- **Hypothesis 1:** Participants will interact more times with the conversational agent interface than with the QA interface.
- **Hypothesis 2a:** Participants will use natural language queries significantly more times than keyword queries to consult the system.
- **Hypothesis 2b:** In particular, participants will use natural language queries slightly more times with the conversational agent interface than with the QA interface.
- **Hypothesis 3a:** Participants will prefer to follow suggestions and ask for related topics significantly more times rather than asking for different topics.
- **Hypothesis 3b:** In particular, participants will prefer to follow suggestions when these are offered so that they have to accept or reject them.
- **Hypothesis 4:** Participants who engage in social dialogue will show significantly more satisfaction in the questionnaire.

The 12 participants that volunteered for this evaluation process were randomly assigned to one of two groups, varying the order in which they will evaluate the two interfaces for counterbalancing (50% of the participants will test the QA configuration first and then the Conversational Agent). Each group receives a evaluation guide and a questionnaire. Each participant is given the task consisting on questioning the system (to add extra motivation we encouraged them to try to trick it or find a relevant topic where to which the system has no answer). They are requested to perform this task with both system configuration. The evaluation consist on: first, the participants fill in a demographics questionnaire; second, they are given the general instructions to follow in the whole evaluation process; then they are given the task and the URL of the system they will test first (according to the group the belong to); after that, they are requested to fill in a questionnaire of satisfaction about the first system configuration; then they are given the URL of the second system to test (and the same task to perform); a second satisfaction questionnaire about the second system configuration is delivered; and finally they are requested to fill in a global satisfaction questionnaire. The average time for performing the whole evaluation process was 9.13 minutes (SD=4.26).

6.4.2 Measurements

During the process, two different measurements were taken: the questionnaires of satisfaction delivered three times during the process, and the interaction measurements taken from the logs resulting from the interaction of each participant with the system. The questionnaires and the interaction measurements were paired using unique session identifiers. After the evaluation process the logs were computed to obtain five metrics: the number of total interactions of each participant with each system configuration, the number of suggestions received by each participant with each configuration, the number of suggestions followed by each participant with each configuration, the format of each participant's query (Natural language or keywords) with each configuration, and the number of participants that interacted using social dialogue with each configuration.

6.4.2.1 Usage of different type of queries

We split the user queries into two groups: those formed of NL and those that formed of keywords. Although in different measure 75% of the participants used a keyword query at least once, and all of them used NL queries. The average number of NL queries per user was 6.79, 1.26 times greater than the average number of keyword-based queries. It is statistically significant ($F_{1,11}$=9.82, p=0.004 $<$ .005) that prefer NL to consult rather than keyword-based queries. This is also verifiable when we split the data of both interfaces. Resulting p values are 0.011 and 0.008 for QA and conversational agent configurations respectively. This support out hypothesis 2a.

However, it is not statistically significant that users use more often NL than keywords with the Conversational Agent than with the QA configuration, so hypothesis 2b cannot be validated. Figure 6.6 shows this distribution, and it can be appreciated that the number of interactions increases with the Conversational Agent configuration for both query types. We concluded this is because, users in general have their own preferences about use natural language or keywords (or possibly influenced by the nature of the interface used) regardless of the nature of the answers received. It is important to point out that only two out of 12 users in the experiment used more often keyword queries than NL queries. Here there exists a subtle group effect, since users from Group B (which interacted first with the conversational agent) slightly reduced the usage of Keyword queries when they interacted with the QA interface.

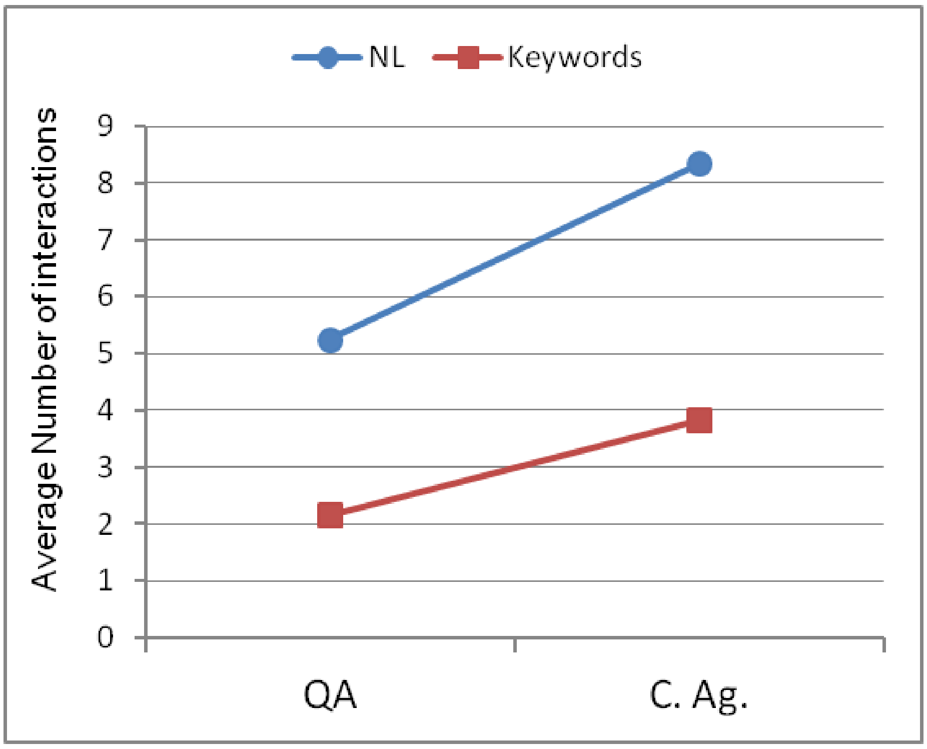


Figure 6.6: Average number of interactions per user using each query type for each system configuration

6.4.2.2 Impact of suggestions

An average of 57% of the times a user was given a suggestion he/she followed that suggestion in the next interaction. Besides, users rated the usefulness of suggestions with 3.41 over 4. However, 96% of the suggestions followed were close-ended questions, e.g. "Do you want to check for *inheritance* too?". Thus we can conclude that users are interested in suggestions when they only need to accept or reject them (hypothesis 3b), but not that they follow suggestions of any type (hypothesis 3a).

6.4.2.3 Satisfaction

According to the results shown in Fig. 6.7, no correlation between interaction metrics and user's satisfaction is appreciated. Thus we cannot validate hypothesis 4.

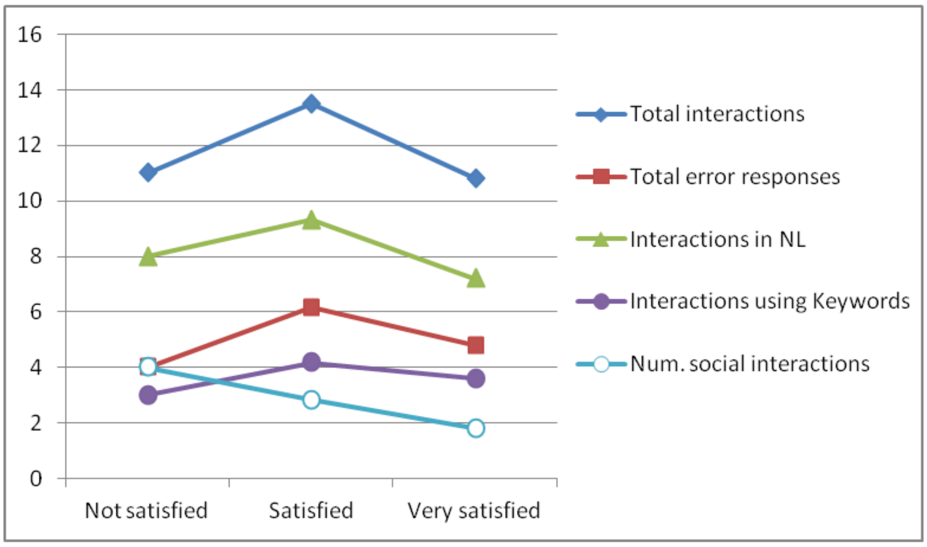


Figure 6.7: User interaction metrics sorted by its satisfaction

6.5 Summary

In this chapter we have analysed the performance of the prototype for the Java programming language.

First, we took a look at the resources the system used, both under low load and for a period of time while we performed multiple stress tests. We also studied the response times of the system under a high number of queries, and analyse the results.

Then, we used a corpus with multiple questions and interactions to automatically test for the success rate of the system, in comparison with each of the modules.

Finally, we described the process followed to test the system with real users, comparing our system with a regular QA system, and presented the results.

CHAPTER 7

Conclusion and future work

In this chapter we present the final results of this Master Thesis, take a look at the goals we started this project with and its completion, to finally propose several possible improvements for future work.

7.1 Conclusion

For this project, we have developed multiple conversational agents that allowed users to interact with Linked Data Systems in Natural Language. We will now present the conclusions we have deduced from our work.

We started this document specifying several requirements and goals we aimed to achieve with the architecture and prototypes developed for this Master Thesis. It has been shown that with the technologies available at this point is possible build personal assistants using conversational agents and Linked Data to help final users find the information they are looking for. Building these systems comes from the appropriate knowledge of this technologies, and how they can communicate with each other.

The Natural Language processing and understanding capabilities of ChatScript have allowed us to present an interface that users with low technical capabilities can use, effectively lowering the entry barrier for the fields we can adapt the system for. However, it is important to remark that Natural Language processing still needs to improve, as can be shown by the results of the evaluation. This is specially true for languages other than English, since most frameworks for natural language processing are aimed at English speaking users, and adapting them for different languages has proved that can be a daunting task if performed in full.

By developing two different prototypes for the proposed architecture, with different types of documents, we have shown that our system can be adapted to different fields with relatively low effort, although it was shown than the first prototype was not ready for an environment formed by heterogeneus documents containing the required data.

The tests with the users have allowed us to check the hypotheses formulated at the start of this project, confirming several of them and being unable to support others.

Finally, the offering of our conversational agent as a web application, has proved a fundamental tool to spread the usage of our system, since it permits access form any device with an Internet connection.

7.2 Achieved goals

In chapter 1 we discussed the goals we wanted to achieve with this project. This can be summarize as follows:

(i) Develop a system that would be able to take a natural language requests and provide answers using a Linked Data System.

(ii) Evaluate different natural language and linked data systems to use with our own.

(iii) Study the implementation of the system in multiple fields of knowledge.

(iv) Analyse the behaviour of the system, as well as the experience of the users using it.

For the first goal, we have proposed and architecture that can fulfil that objective, as well as developed two prototypes proving it was possible to successfully follow that architecture while developing a system with a Natural Language Interface.

We studied the systems for Natural Language processing available, analysing their capabilities and their weaknesses, focusing on the possibilities of using languages other than English for the processing. We also analysed multiple Linked Data systems, as well as Linked Open Data systems, finally choosing the one we judged the most appropriate for our system. Therefore, we consider the goal (ii) to have been reached.

The first prototype developed for our system was focused on providing assistance for an e-learning platform for the Java programming language, focusing on very simple and structured documents, therefore not presenting significant challenges to index. We then developed a system focused on finding information about a research group, with documents of different structures and formats, dealing with different topics. We achieved this by extending our first system into a new one capable of handling these new requirements. Accordingly, we consider goal (iii) to be achieved.

Finally, we proposed a methodology to test the system, and test whether our hypotheses about user experience and behaviour were correct. We perform an experiment comparing our system with a QA interface, with real users, and gathering data from their usage of the system, leading to the publication of the findings [26]. We also performed tests to check the performance of the system under different loads, showing the final results in chapter 6. These tests achieved goal (iv).

7.3 Future work

Working on the systems developed for this Master Thesis, we have achieved an understanding of the possibilities of Natural Language Processing, as well as the benefits of the Semantic Web and the Linked Open Data initiative. With this knowledge, we can suggest future areas of study that can improve the functionality of the project and extend its goals:

1. *Spanish dictionaries*: Although it is possible to use the Chatbot engine chosen for our prototypes, ChatScript, with Spanish, it has a distinct lack of support, and nowhere near the capabilities for the English language. To improve this, it would be necessary to build Spanish dictionaries, or translate the existing ones from English, as well as analyse the processing ChatScript does for English sentences and mimic it with Spanish, adapting it when necessary.

2. *Using SPARQL*: integrating a SPARQL query system would allow to easily adapt our system for multiple knowledge fields, as well as providing support for more complex interactions with the user, given the huge capabilities of the Linked Data.

3. *Automating the indexing process*: Although at this point the scrapping process is mostly automated and can be done without human interaction, it is still necessary for a person to gather the scrapped data and index it in Solr. In order to make our system work in semi-real time, it would be necessary to fully automate the process, from launching the multiple scrappers to combining the information and adding it to the system, be it Solr or a SPARQL server.

4. *Extend the evaluation experiment to the second prototype*: The described experiment with the users was performed with the first prototype. Adapting the methodology to the second one, and gathering the users feedback may be useful for further improvements of the system.

5. *Adding audio interface*: Whilst we consider interacting with our system with natural language an improvement in comparison with traditional interfaces, it could be further improved by including support for speech recognition and text-to-speech capabilities, simplifying user interactions, and greatly improving accessibility.

6. *Administrative interface*: Adding an interface to handle the different modules of the system, as well as checking their status and success rate in real time will allow administrators to easily manage the systems and respond to any possible problem that may appear.

7. *Responsive web client*: Given the current limitations of html and iframes, our system does not behave well in smaller screens. Trying to overcome this limitation will greatly improve the user experience when using our system.

In conclusion, we have developed a robust and functional system, but it can still be greatly improved with new modules, enhancing the user experience. As new ideas and new developers come forward, they could take this system as a base to develop better and more powerful personal assistants.

APPENDIX A

Installation Manual

This chapter aims to provide a general walk-trough on how to deploy our prototypes bot on a fresh system. It assumes basic knowledge on both the system and the tools used, and the required tools.

A.1 Order of deployment

Our system is composed of several submodules, and it is reccomended that they are started in a specific order:

1. Chatscript
2. Apache Solr
3. Front-end controller
4. Front-end client

A.2 Chatscript

Before being able to launch ChatScript, if you are using a 64 bit system, you need to install several 32-bit libraries, as shown in A.1:

```
:~/$ sudo dpkg --add-architecture i386
:~/$ sudo apt-get update
:~/$ sudo apt-get install lib32gcc1 lib32stdc++6
```

Listing A.1: Libraries for ChatScript

For the first launch of Chatscript, you need to compile the corpus. Therefore, you have to launch chatscript on localmode first, issue the build commands, and then launch it as a server. The commands are shown in A.2 for the build, and, once de process is complete, exit the local interface with ":quit", and launch it as a server, as shown in A.3:

```
:~/calista-bot/ChatScript$ ./LinuxChatSscript64 local
[...]
Enter user name: username
username: > :build Dent
[...]
username: > :quit
```

Listing A.2: ChatScript build command

Keep in mind that, even though we refer to our bot as "Duke", the files for this project are the ones from the "Dent" folder, since the existing "Duke" files are related to a previous iteration and where keep independent.

```
  :~/calista-bot/ChatScript$ ./LinuxChatScript64 port=1025
arg[1] = port=1025
evserver child fork requests: 0
  evserver: running
Server EVSERVER ChatScript Version 5.1  64 bit LINUX compiled Feb  1 2015
    23:00:53
EVSERVER ChatScript Version 5.1  64 bit LINUX compiled Feb  1 2015 23:00:53
Params:   dict:720895 fact:800000 text:35000kb hash:50000
          buffer:22x80kb cache:1x50kb userfacts:100
```

```
WordNet: dict=190271  fact=84865  stext=12742900 Jan31'15-15:26:50
Build0:  dict=68775  fact=130830  dtext=1162676 stext=0 Feb01'15-15:07:58
    0.txt
Build1:  dict=8  fact=7  dtext=356 stext=3664 Jun27'15-22:42:15 Niko.txt
Error(2) opening LIVEDATA/ENGLISH
Used 49MB: dict 259,059 (24869kb) hashdepth 17/3 fact 215,702 (8628kb) text
     13910kb
           buffer (1760kb) cache (50kb) POS: 0 (0kb)
Free 44MB: dict 461,836 hash 1136 fact 584,298 text 21,089KB

    Dictionary building disabled.
    Postgres disabled.



======== Began EV server 5.1 compiled Feb  1 2015 23:00:53 on port 1025 at
    Tue Jun 30 23:41:37 2015 serverlog:1 userlog: 0


======== Began EV server 5.1 compiled Feb  1 2015 23:00:53 on port 1025 at
    Tue Jun 30 23:41:37 2015 serverlog:1 userlog: 0
evserver: parent ready (pid = 12435), fork=0
EVServer ready: LOGS/serverlog1025.txt
```

Listing A.3: ChatScript Server running

Keep in mind it will bind to the 1025 port.

A.3 Front-end controller

The controller need two libraries to be installed with pip. It is recommended to do it in a virtualenvironment, as shown in A.4.

```
  :~/$ mkvirtualenv askbot
  (askbot):~/$ pip install websocket-client unidecode
```

Listing A.4: Libraries for the front end controller

The askbot controller binds, by default, to the 4242 port, although it can be changed in the askbot.py file. Once the changes have been made, it can just be run with python as in A.5

```
(askbot):~/calista-bot/FE-Controller$ python askbot.py
```

Listing A.5: Running the front end controller

A.3.1 Using apache mod_wsgi

Using the provided askbot.wsgi, the system can be run using Apache and mod_wsgi. After installing and enabling the mod (in Debian systems, see listing A.6, set the virtualenvironment in the configuration files, as well as the appropriate Aliases, as shown in A.7

```
(askbot):~/$ sudo apt-get install libapache2-mod-wsgi
(askbot):~/$ sudo a2enmod wsgi
(askbot):~/$ sudo apache2ctl restart
```

Listing A.6: Installing mod_wsgi

```
# Run with the virtual environment
WSGIDaemonProcess talkbot python-path=/path/to/virtualenvsfolder/VEnvs/
   talkbot/lib/python2.7/site-packages

# The path for the .wsgi
WSGIScriptAlias /AskBot /path/to/calista-bot/FE-Controller/askbot.wsgi
<Location /AskBot>
    WSGIProcessGroup talkbot
    Order deny,allow
    Allow from all
</Location>
```

Listing A.7: Apache WSGI configuration

A.4 Front-end client

The client for the bot is a web application. You just need to place the Ask-client files on a webserver (like Apache or Nginx) html folder, and edit the index.html file, so the action attribute in the form points to the host and port from the previous step, or to the Apache alias is set up with mos_wsgi.

A.5 Apache Solr

Apache Solr can be run in a Tomcat installation. You will need to indicate the solr.home in a context specific for Solr, where the schemas and the Solr libraries are located. An example context is shown in A.8. This file should be stored inside the "conf/Catalina/localhost" folder, and with the same base name as the .war. For example, for an application solr.war, the appropriate context file will be solr.xml

```xml
<Context docBase="/Path/to/tomcat/Tomcat/webapps/solr.war" debug="0"
    crossContext="true">
  <!-- The path to the folder where the solr installation is located
      Production environments should avoid using the example Solr instance
           -->
  <Environment name="solr/home" type="java.lang.String" value="/home/
      amardomingo/PFC/Solr/example/solr" override="true"/>

  <!-- Allow access to Solr only from localhost -->
  <Valve className="org.apache.catalina.valves.RemoteHostValve" allow="
      localhost"/>
</Context>
```

Listing A.8: Tomcat context for solr

APPENDIX B

Solr Uploader

The complete RDF scrapped for this document is avaliable in `http://github.com/gsi-upm/calista-bot`, since the files are too long to be included here. We will show, however, the uploader developed to index the files in solr:

B.1 Command syntax

The uploader can handle rdf and json files, and takes the arguments explained in table B.1 . And example usage is shown in listing B.1

```
:~/calista-bot/RDF$ ./uploader.py -u http://localhost:8080/solr -c
    elearning -r vademecum.rdf -e
```

Listing B.1: Example command to upload the vademecum data into Solr

Argument	**Short argument**	**Parameter**	**Explanation**
–help	-h		Shows the help
–url	-u	URL	Sets URL as the Solr endpoint
–core	-c	CORE	Uses CORE as the Solr core to upload the data to
–data	-d	JSONFILE	Reads the json data in the JSONFILE file to upload it to solr
–rdf	-r	RDFFILE	Reads the rdf data in the RDFFILE file to upload it to solr
–verbose	-v		Prints debug information
–empty	-e		Clears all the data in solr before uploading any document
–output	-o		Logfile to output to

Table B.1: Parameters for the uploader to Solr.

B.2 Uploader code

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

from __future__ import print_function

import sys
import argparse

import requests

from xml.dom import minidom
import json

solr_url="{url}/{core}/update"
solr_commit="<commit/>"
solr_clear="<delete><query>*:*</query></delete>"
solr_charset ='utf-8'

def get_tags():
    """
    Read the taglist file for the equivalences between rdf and json
        structure
    """
    flines = open('taglist.txt').readlines()
    eq = {}
    for line in flines:
        if line != '':
            line_data = line.split(',')
            eq[str(line_data[0])] = str(line_data[1].strip())
    print(eq)
    return eq



def commit(args):
    '''
    Commits the changes
    '''
    url = solr_url.format(url=args.url, core=args.core)
    headers = {'Content-type':'text/xml', 'charset':solr_charset}
    requests.post(url, data=solr_commit, headers=headers)
```

```
def clear(args):
    '''
    Clears the solr core
    '''
    url = solr_url.format(url=args.url, core=args.core)

    headers = {'Content-type':'text/xml', 'charset':solr_charset}

    requests.post(url, data=solr_clear, headers=headers)
    commit(args)

def read_xml(data_file):
    """
    Read the data from the xml file
    """
    # Correspondence
    rdf2json = get_tags()

    xmlfile = minidom.parse(data_file)

    # Each element in this file, is an rdf:Description
    itemlist = xmlfile.getElementsByTagName('rdf:Description')

    documents = []
    for element in itemlist:
        doc = {}
        # First, read all this attributes
        for node in element.attributes.items():
            if node[0] in rdf2json:
                doc[rdf2json[node[0]]] = node[1]
        # Now, repeat for the children

        for childnode in element.childNodes:
            name = str(childnode.nodeName.strip())
            if name in rdf2json:
                value = ""
                # the value for rdf:type is sometimes stored as an
                    attribute
                if len(childnode.attributes.items()) != 0:
                    value = childnode.attributes.items()[0][1]
                else:
                    value = childnode.firstChild.nodeValue
                doc[rdf2json[name]] = value

        # The doc is complete
        documents.append(doc)
    return documents

def read_json(data_file):
    '''
    Read the data from the provided json
    '''

    json_file = open(data_file, 'r')

    data = json.loads(json_file.read())

    return data

def upload_doc(doc, args):
    '''
    Upload a doc to solr
    '''
    url = solr_url.format(url=args.url, core=args.core)
    operations = {'add':{'doc':doc}}

    headers = {'Content-type':'application/json', 'charset':solr_charset}

    requests.post('{url}/json'.format(url=url), data=json.dumps(operations)
        , headers=headers)

def main(args):
    '''
    Perform the necessary requests
    '''
    # Clears the core, if asked
    if args.empty:
        clear(args)
        if args.verbose:
            print("Data in core {core} cleared".format(args.core), file=
                args.output)

    if args.verbose >2:
        print("Reading data from {file}".format(args.file), file=args.
            output)
    data = []
    if args.data:
        data += read_json(args.data)
    elif args.rdf:
        data += read_xml(args.rdf)
    else:
        print("Need at least an rdf or json file to read", file=sys.stderr)
        exit(1)

    # There is no actual numeric id, so...
    i = 1
    for doc in data:
        if 'id' not in doc:
            doc['id'] = i
        i+=1
        upload_doc(doc, args)
    commit(args)

    print("Uploaded all {number} doc to solr".format(number=str(len(data)))
        , file=args.output)

if __name__=='__main__':
    parser = argparse.ArgumentParser(description="Uploader for solr",
        add_help=True)
    parser.add_argument('-u', '--url', default="http://localhost:8080/solr"
        , help="URL for the solr install")
    parser.add_argument('-c', '--core', default="gsidata", help="The core
        in use")
    parser.add_argument('-d', '--data', default="gsisemanticdata.json",
        help="A file with json data")
    parser.add_argument('-r', '--rdf', default=None, help="A file with rdf
        data")
    parser.add_argument('-v', '--verbose', action='count', help="Print
        debug info")
    parser.add_argument('-e', '--empty', action='store_true', help="Clear
        the data before upload")
    parser.add_argument('-o', '--output', default=sys.stdout, help="Log
        output file")
    parser.set_defaults(empty=False)
    args = parser.parse_args()
    # Get log output
    if args.output != sys.stdout:
        args.output = codecs.open(args.output, 'w+', 'utf-8-sig')
    main(args)
```

Listing B.2: Uploader to index the scrapped data in solr

Bibliography

[1] F. A. M. Fonte, J. C. Burguillo, and M. L. Nistal, "An intelligent tutoring module controlled by bdi agents for an e-learning platform," *Expert Systems with Applications*, vol. 39, no. 8, pp. 7546–7554, 2012.

[2] A. Bogdanovych, K. Ijaz, and S. Simoff, "The city of uruk: Teaching ancient history in a virtual world," in *Intelligent Virtual Agents* (Y. Nakano, M. Neff, A. Paiva, and M. Walker, eds.), vol. 7502 of *Lecture Notes in Computer Science*, pp. 28–35, Springer Berlin Heidelberg, 2012.

[3] A. Augello, G. Pilato, G. Vassallo, and S. Gaglio, "A semantic layer on semi-structured data sources for intuitive chatbots," in *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference on*, pp. 760–765, IEEE, 2009.

[4] R. Wallace, N. Bush, T. Ringate, A. Taylor, and J. Baer, "Airtifical intelligence markup language version 1.0.1 - A.L.I.C.E. AI foundation working draft." `http://www.alicebot.org/TR/2001/WD-aiml/`, 2001. Accessed: 2015-06-12.

[5] "AIML: Artificial intelligence markup language." `http://www.alicebot.org/aiml.html`. Accessed: 2015-06-12.

[6] R. S. Wallace, "Aiml 2.0 working draft." `https://docs.google.com/document/d/1wNT25hJRyupcG51aO89UcQEiG-HkXRXusukADpFnDs4/pub`.

[7] B. Wilcox and S. Wilcox, "Suzzete, the most human computer," 2010.

[8] B. Wilcox, "Chatscript." `http://sourceforge.net/projects/chatscript/`. Accessed: 2015-06-17.

[9] C. Unger, A. Freitas, and P. Cimiano, "An introduction to question answering over linked data," in *Reasoning Web. Reasoning on the Web in the Big Data Era* (M. Koubarakis, G. Stamou, G. Stoilos, I. Horrocks, P. Kolaitis, G. Lausen, and G. Weikum, eds.), vol. 8714 of *Lecture Notes in Computer Science*, pp. 100–140, Springer International Publishing, 2014.

[10] P. Cimiano, P. Haase, J. Heizmann, M. Mantel, and R. Studer, "Towards portable natural language interfaces to knowledge bases - the case of the orakel system," *Data and Knoledge Engineering*, vol. 65, no. 2, pp. 325–354, 2008.

[11] C. Unger and P. Cimiano, "Pythia: Compositional meaning construction for ontology-based question answering on the semantic web," in *Natural Language Processing and Information Systems* (R. Muñoz, A. Montoyo, and E. Métais, eds.), vol. 6716 of *Lecture Notes in Computer Science*, pp. 153–160, Springer Berlin Heidelberg, 2011.

[12] V. Lopez, M. Fernández, E. Motta, and N. Stieler, "Poweraqua: Supporting users in querying and exploring the semantic web," *Semantic Web*, vol. 3, no. 3, pp. 249–265, 2011.

[13] V. Lopez, V. Uren, E. Motta, and M. Pasin, "Aqualog: An ontology-driven question answering system for organizational semantic intranets," *Web Semantics: Sience, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 72–105, 2007.

[14] C. Unger, L. Bühmann, J. Lehmann, A.-C. Ngonga Ngomo, D. Gerber, and P. Cimiano, "Template-based question answering over rdf data," in *Proceedings of the 21st international conference on World Wide Web*, pp. 639–648, ACM, 2012.

[15] A. Freitas, J. G. Oliveira, S. O'Riain, E. Curry, and J. C. P. Da Silva, "Querying linked data using semantic relatedness: a vocabulary independent approach," in *Natural Language Processing and Information Systems*, pp. 40–51, Springer, 2011.

[16] G. Ingersoll, T. Morton, and A. Farris, *Taming Text: How to Find, Organise, and Manipulate it.* Manning Pubs Co Series, Manning, 2013.

[17] J. I. Fernández-Villamor, C. A. Iglesias, and M. Garijo, "A Framework for Goal-Oriented Discovery of Resources in the RESTful Architecture," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, pp. 796–803, June.

[18] T. Berners-Lee, "Information management: A proposal," 1989.

[19] T. A. S. Foundation, "Apache http server project." `http://httpd.apache.org/ABOUT_APACHE.html`. Accessed: 2015-06-22.

[20] P. J. Eby, "Python web server gateway interface v1.0," Tech. Rep. PEP 333, December 2003.

[21] P. J. Eby, "Python web server gateway interface v1.0.1," Tech. Rep. PEP 3333, September 2010.

[22] B. Samei, H. Li, F. Keshtkar, V. Rus, and A. C. Graesser, "Context-Based Speech Act Classification in Intelligent Tutoring Systems," in *Intelligent Tutoring Systems*, pp. 236–241, Springer, 2014.

[23] C. Moldovan, V. Rus, and A. C. Graesser, "Automated speech act classification for online chat," *CEUR Workshop Proceedings*, vol. 710, pp. 23–29, 2011.

[24] "Flask, a uwsgi framework for python." `http://flask.pocoo.org/`. Accessed: 2015-05-14.

[25] "Wsgi for apache." `https://github.com/GrahamDumpleton/mod_wsgi`. Accessed: 2015-05-14.

[26] M. Coronado, C. Iglesias, and A. Mardomingo, "A personal agents hybrid architecture for question answering featuring social dialog," in *2015 IEEE International Symposium on Innovations in Intelligent Systems and Applications (INISTA 2015)*, (Madrid, Spain), September 2015.