SEVENTH FRAMEWORK PROGRAMME

IST FP7 231507

D4.7 - Release of Open Source Framework V3.0

File name	PuppyIR-D4.7-Release-of-Open-Source-Framework-V3.0
Author(s)	José Miguel Garrido (ATOS) Diego Esteban (ATOS) Paul Moore (ATOS) Douglas Dowie (UGLW) Richard Glassey (UGLW) Leif Azzopardi (UGLW) Karl Gyllstrom (KUL)
Work package/task	WP4
Document status	Final
Contractual delivery date	M36
Confidentiality	Public
Keywords	Framework Implementation and Release
Abstract	Release notes to accompany the release of the third and final version of the PuppyIR Framework.

EXECUTIVE SUMMARY **3**
1 INTRODUCTION **4**
1.1 AUDIENCE 4
1.2 IMPLICATIONS FROM D4.5 – REPORT ON SPECIFICATION AND DESIGN CHANGES 4
1.3 REPORT STRUCTURE 4
2 FRAMEWORK MAP **5**
2.1 TRUNK 5
2.1.1 Framework 6
2.1.2 Trunk/Demonstrators 6
2.1.3 Prototypes 9
2.1.4 Interfaces 9
2.2 BRANCHES 9
2.2.1 Work-in-progress prototypes 11
2.3 TAG 11
3 ABOUT THE FRAMEWORK **13**
3.1 OVERVIEW AND BACKGROUND 13
3.2 THE MAIN FEATURES OF THE FRAMEWORK 13
3.2.1 Data Formatting 13
3.2.2 Architectural Paradigms 14
3.2.3 Event and Query Logging 14
3.2.4 Query and Result Filters 14
3.2.5 Query and Result Modifiers 15
3.2.6 Search Engine Wrappers 15
3.3 EXTENSIBILITY OF THE FRAMEWORK 17
3.3.1 Search Engine Wrappers 17
3.3.2 Query and Result Filters &Query and Result Modifiers 17
3.4 PARADIGM 1: ONE PIPELINE, ONE SEARCH ENGINE 18
3.4.1 Description of the components 18
3.4.2 Data flow in the 'Service' paradigm 18
3.4.3 The Query and Results formats 19
3.5 PARADIGM 2: ONE PIPELINE, MANY SEARCH ENGINES 19
3.5.1 Description of the components 20
3.5.2 Data flow in the 'Pipeline' paradigm 20
3.5.3 The Query and Results formats 21
3.5.4 Possible advantages of using this architecture 22
4 DEVELOPMENT ROADMAP **23**
4.1 INTEGRATION OF PUPPYIR AND DJANGO 23
4.2 CONFIGURATION OF PUPPYIR SERVICES 24
5 SUMMARY **25**
6 ANNEXES **26**
6.1 ANNEX A: INSTALLING THE FRAMEWORK 26
6.1.1 Requirements and Installation 26
6.1.2 Other features and framework support 29
6.1.3 Standalone Services 29
6.1.4 Proxy Server Support 29
6.2 ANNEX B: USING THE FRAMEWORK 31

6.2.1 Prototypes *31*
6.2.2 Building a Standalone PuppyIR Service *37*
6.2.3 Exception Handling in PuppyIR *40*
6.2.4 The PuppyIR Framework Test Suite *43*
6.3 ANNEX C: TUTORIALS 45
6.3.1 BaSe Tutorial: Building a PuppyIR/Django Service *45*
6.3.2 IfSe Tutorial: Information Foraging Search Application *48*
6.3.3 MaSe Tutorial: Mash-up Search Engine Application *52*
6.3.4 Pipeline Tutorial: DeeSe (Detective Search) *58*
6.3.5 Tutorial: Configuration editor *61*
6.4 ANNEX D: EXTENDING THE FRAMEWORK 67
6.4.1 Extending the Query Pipeline *67*
6.4.2 Extending the Result Pipeline *69*
6.4.3 Adding new Search Engine Wrappers *71*
6.4.4 On Filters, Modifiers and Query Logging *74*

REFERENCES 76

Executive Summary

This report accompanies the third and final major release of the PuppyIR Framework. These release notes provide: a map of the framework development activities (including details of the extensive revisions to the framework); a discussion of the main features of the framework; a look back at the requirements [2, 3, 8, 9] with relation to what has been achieved, and as a technical annex, a summary of the tutorials developed for the framework and an installation guide.

This deliverable builds upon the previous WP1, WP2, WP3, WP4 and WP7 deliverables:

- D1.2 – Agreed User Requirements and Scenarios [2]
- D1.3 – Agreed Technical Requirements [3]
- D3.1 – Report on Data Pre-processing [4]
- D4.1 – Specification Report [5]
- D4.2 – Design Report [6]
- D4.3 – Report on Implementation and Documentation [7]
- D4.4 – Release of the Open Source Framework V1.0 [8]
- D4.5 – Report on Design and Specification Changes [9]
- D4.6 – Release of Open Source Framework V2.0 [11]
- D7.3 – Hospital Demonstrator – Version 1.0 [12]
- D7.4 – Hospital Demonstrator – Version 2.0 [13]

The source code can be accessed from the project's SVN repository hosted on SourceForge at:

http://sourceforge.net/projects/puppyir/.

The documentation can be accessed online at the following address:

http://www.puppyir.eu/framework

PDF versions of the documentation are also available from the project's SourceForge page at:

http://sourceforge.net/projects/puppyir/files/doc/

1 Introduction

The PuppyIR Project aims to facilitate the creation of child-centred information services, based on the understanding of the behavior and needs of children [1]. The goals of this report are to:

1. Provide a map of the framework contents and its status;
2. Outline of the main features of the framework;
3. Discuss the changes and evolution of this, final, version of the framework.
4. Introduce and discuss the [extensively] revised documentation for the framework;
5. Describe the dependencies and installation process, for using it;

This report serves 3 key purposes:

1. Provide release notes to accompany the source code for the third version of the framework for project developers;
2. Describes the current framework features;
3. Communicate to all project members and stakeholders how the project progressed from its earlier releases towards this final version.

1.1 Audience

This deliverable is intended to be read by both members and developers of the PuppyIR project to communicate the current status of the framework development. It is also intended to be read by third party developers who are planning on using the framework; serving as a companion piece to the frameworks documentation (as made available on the project's website and on SourceForge).

1.2 Implications from D4.5 – Report on Specification and Design Changes

D4.5 [9] reported the changes made to the original specification and design reported in earlier deliverables, D4.1 – Specification Report [5] and D4.2 – Design Report [6]. These changes are reflected in the framework code that has evolved from the first release into the second release. Since D4.5 there have been no major design or specification changes that need to be reported and instead the framework code base has been mostly refactored to improve the comprehensibility and clarity for third party developers.

1.3 Report Structure

The report is divided into the following major sections:

- Section 2 – Framework Map
- Section 3 – About the Framework
- Section 4 – Development Roadmap
- Section 5 – Summary
- Section 6 – Annexes
- References

2 Framework Map

The PuppyIR repository is organized, into three folders: trunk, branches and tags. Each of these folders is detailed below with a picture of their structure and a short description of the key parts contained within them.

To checkout the whole repository (this is a large download of ~600MB) and browse to the top level of the repository use the following commands:

```
$ svn co https://puppyir.svn.sourceforge.net/svnroot/puppyir puppyir
$ cd puppyir
```

N.B. the diagrams shown in this section are simplified, in that, except for a couple of exceptions, no files are shown; only folders. Also, standard Django application folders (like *'site_media'* - for example) are not shown in order to make the diagrams easier to read.

2.1 Trunk

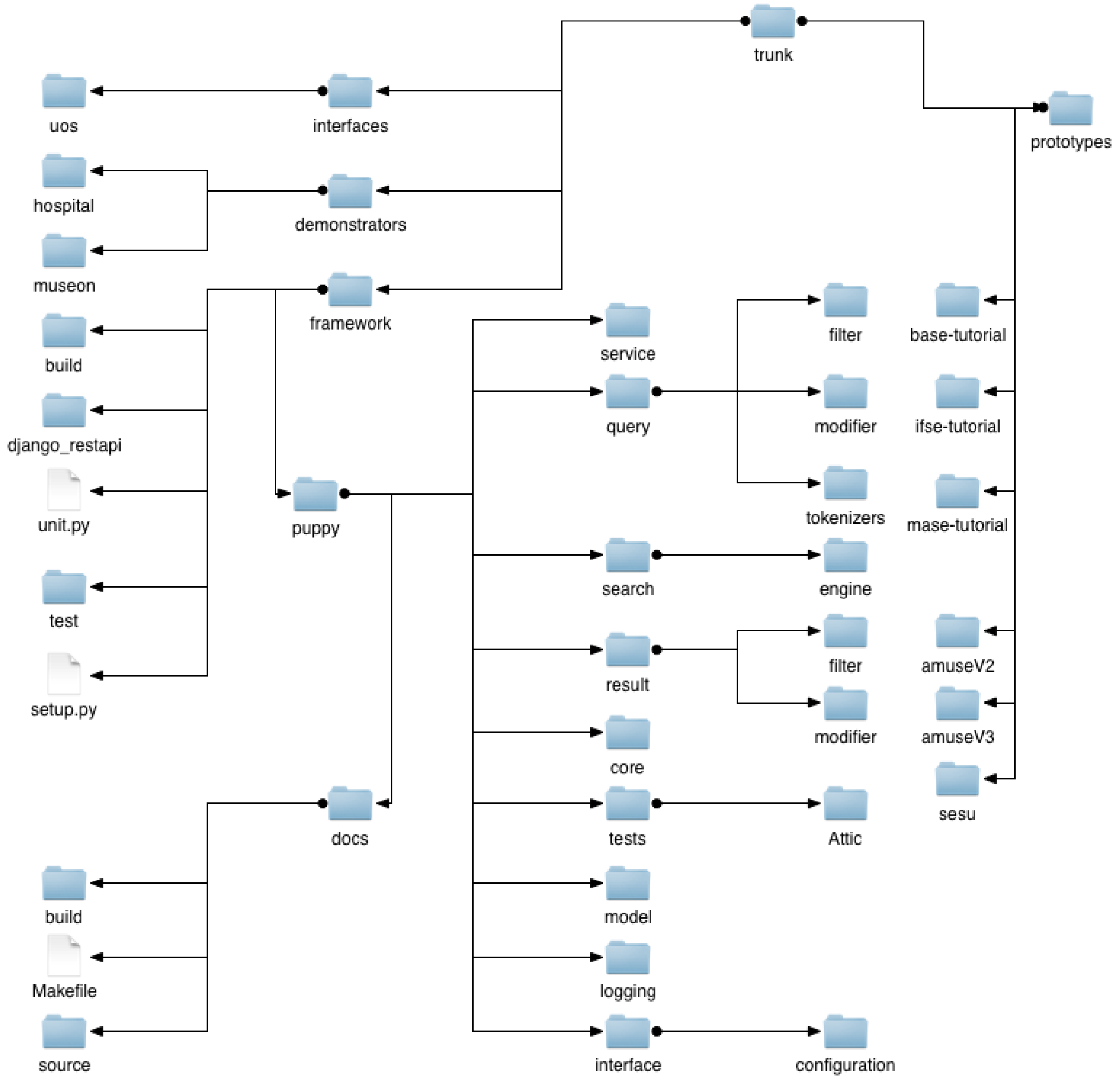

Figure 1 - *Diagram showing the structure of the 'trunk' folder in the repository.*

This section is the main development area of PuppyIR, it contains the latest version of the framework and various applications (plus demonstrators) that make use of it. Following the diagram below, the key sections of the trunk's contents are summarised.

2.1.1 Framework

This folder contains the latest version of the framework, the test suite and the documentation (both the source and 'compiled' versions). The main sections of this part are summarised below in terms of their contents:

- **build and setup.py**: are the build directory (for when installing the framework) and the Python script to install the framework.
- **puppy:** the framework itself, its components are detailed below.
  - **core:** contains a type checking system and also various components for running threads.
  - **docs:** the documentation for the framework, including the source and compiled versions in addition to a make file to build the source.
  - **interface:** contains an early version of a Django application for configuring a search service.
  - **logging:** contains the query and event loggers.
  - **misc**: contains assorted files regarding aspects like stylistic conventions for code in the framework.
  - **model:** contains all the classes associated with the OpenSearch standard.
  - **query:** contains all the filters and modifiers belonging to the query pipeline in addition to the associated exceptions. It also contains various query tokenizers.
  - **result:** contains all the filters and modifiers belonging to the result pipeline in addition to the associated exceptions.
  - **search:** contains all the search engine wrappers and associated exceptions.
  - **service:** contains the service manager and search service classes. It also contains early work on configurable versions of the aforementioned, but, since these are tied into Django - they are not automatically imported by the framework.
  - **tests:** an old version of the test suite; it is an example of low level testing, writing the test programs from scratch. It should be seldom necessary to use this old version style. The new, more user friendly version is detailed below and supersedes this one.
- **test and unit.py:** contains the test suite directory and the Python script for running the tests.

2.1.2 Trunk/Demonstrators

In the trunk there are two demonstrators which serve as showcases for the PuppyIR project; these demonstrators are described below.

2.1.2.1 Hospital Demonstrator [12, 13]

This demonstrator, called as the Emma Search service (EmSe), is being developed for Emma Kinderziekenhuis (EKZ). Mrs Kinderziekenhuis is part of the Amsterdam Medical Centre (AMC).

At the EKZ, children have access to a dedicated information centre as well as a dedicated bedside terminal. A case study carried out by hospital staff from the information centre has revealed that children are reluctant to engage with the physical information centre (depending on if it is the case of a family member or a carer). EmSe is designed to make use of these bedside terminals to allow them to access this resource via the web.

The main goals of this demonstrator are:

- To improve knowledge of existence and possibilities of the information centre;
- To improve the accessibility of the information centre and its content for children;
- To expand the information content with reference to more extensive information on the internet that is both appropriate and suitable.

EmSe assists the children by providing appropriate query suggestions, simplifying difficult content and filtering un-suitable content for children.

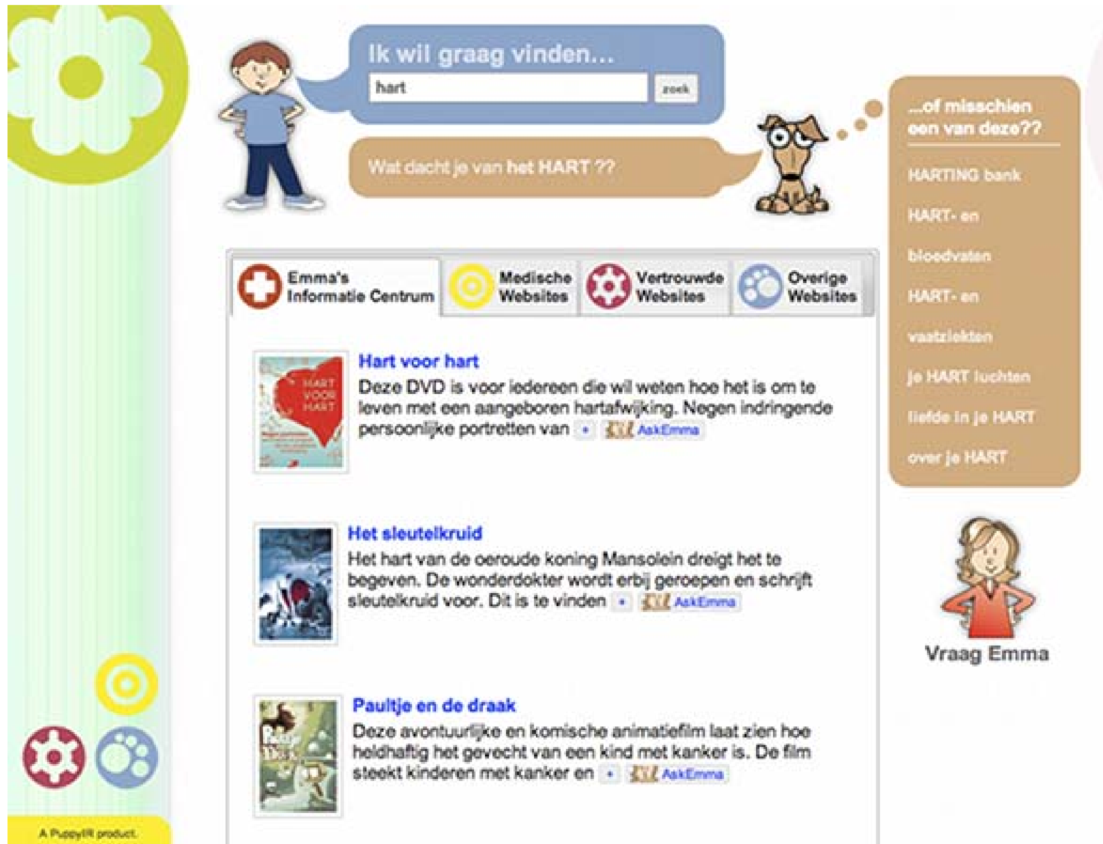

Figure 2 - *EmSe in action showing results from all the services; the dog's speech bubble is a query suggestion with the thought bubble containing more suggestions.*

2.1.2.2 Museon Demonstrator [14]

The Museon Demonstrator creates an interactive museum visit using advanced technologies such as multi-touch tables and marker tracking. The interactive application for multi touch tables does not need to make use of the Open Search Framework, but it is the basis for additional data retrieval and filtering using the post-visit application (which applies the Open Source Framework).

Up to four users can use a multi-touch table simultaneously to browse through the different exhibition subjects and together they determine the contents of an interactive quest. Subsequently in a trail through the exhibitions users/players answer questions related to the chosen topics that have to be found. Throughout the museum various touch-screens equipped with scanners are installed for reading and identifying the player's ticket.

Using tickets as identifications, the touch screens present customized questions and provide feedback to answers. After all questions have been answered, the multi-touch table provides further information about the visited exhibits.

Another application, the post visit web application is available for additional information at home or school. This application can be used outside the Museon premises for additional

search about the subjects chosen in the visit, using again the ticket as identification. The post-visit web page makes use of the Open Search Framework for searching and filtering the results.

The Museon demonstrator is more deeply described in [14] D7.2 'Museon demonstrator version 2.0'

Figure 3 - *The Museon demonstrator being used on multi-touch tables; showing various topics.*

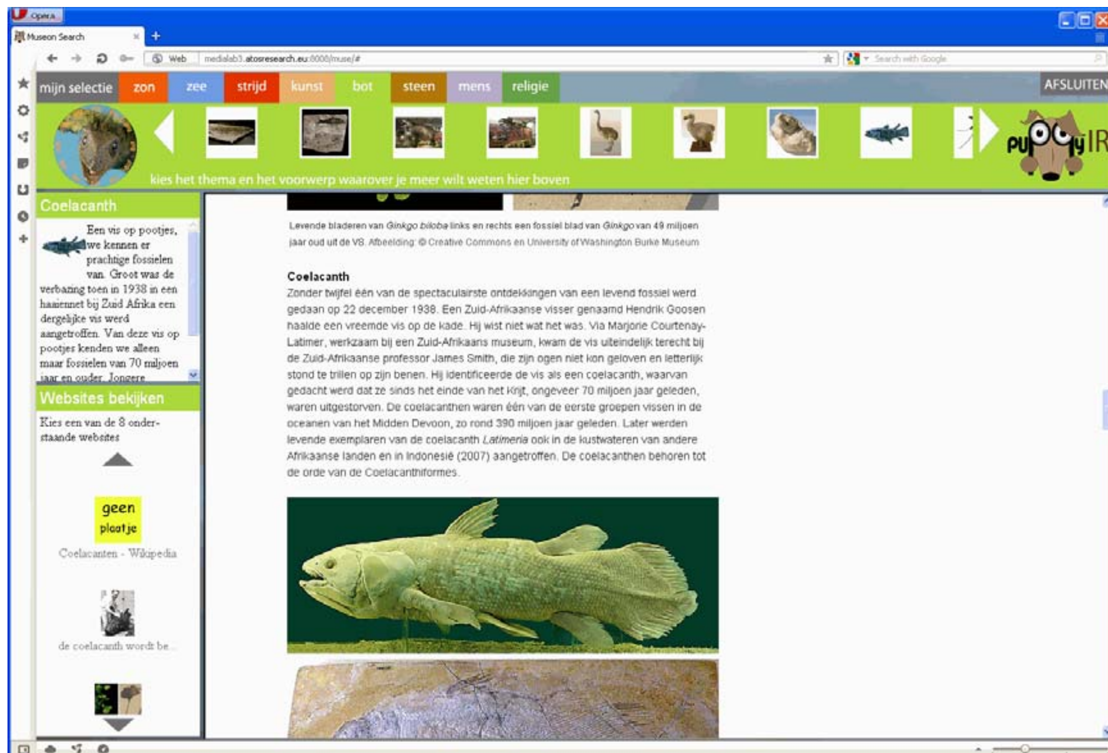

Figure 4 - *The Post-Visit web page, showing additional information about a topic chosen in the visit.*

2.1.3 Prototypes

This folder contains prototypes made using the latest version of the framework. These prototypes are either completed or in the late stages of development and so are all in a demonstrable state.

These prototypes are explained later in this report.

2.1.4 Interfaces

This folder contains the experimental environment on collaborative search interfaces.

2.2 Branches

This folder contains standalone components and unfinished/work-in-progress prototypes.

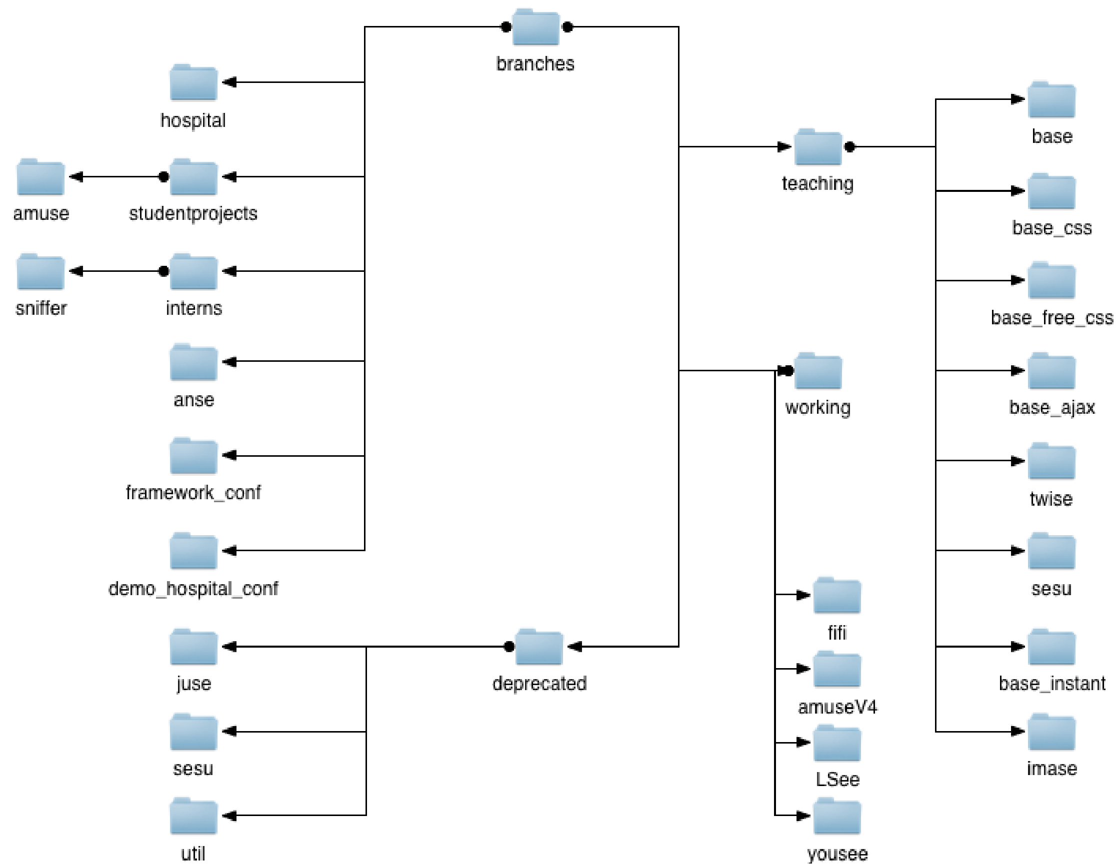

Figure 5 - *Diagram showing the structure of the 'branches' folder in the repository.*

Branches node contains:

- **AnSe** this is an application that uses the PuppyIR framework to query, using the Bing and YouTube wrappers, and retrieve results in the JSON format. It is totally standalone as it contains its own, reduced, local copy of the PuppyIR framework.
- **conf demos (framework and hospital)** these are early versions of a method to allow for easy configuration of these resources.
- **Interns**: an application called 'sniffer' created by student interns working on PuppyIR, this application consists of: a search application similar to BaSe (see below for more on BaSe) and an automated logging application called ALF (Automated Logging Facility).
- **Student projects** this contains applications made by students studying the *Internet Technology* module at the University of Glasgow. At present it only contains the original version of the aMuSe application (the new versions, as detailed earlier, can be found in trunk) using an old version of the PuppyIR framework.
- **Teaching:** this folder contains various applications created (using the PuppyIR framework) as part of the *Internet Technolnology and Distributed Information Management courses at the University of Glasgow.*
  - **BaSe:** a basic search engine that searches for and display web results.
  - **BaSe CSS:** same as BaSe but with CSS styling applied to it.
  - **BaSe Free CSS:** same as BaSe but with multiple different styles available and style switching code (in JavaScript).
  - **BaSe Ajax:** same as BaSe but it searches for, retrieves and displays web results using Ajax.
  - **BaSe Instant:** same as above but using code from a live in-lecture demo - no major differences to BaSe Ajax.
  - **TwiSe:** a basic twitter search engine for finding and displaying tweets.
  - **SeSu:** another alternate version of the now deprecated SeSu prototype.
  - **ImaSe:** a basic image search engine for finding and displaying images.

- **Working:** this folder contains prototypes that, while using the latest version of the framework, are still work-in-progress.
- **Deprecated**: these prototypes use an outdated local version of the framework (called 'util'). SeSu does not work anymore but JuSe does still function. Both applications and 'util' are no longer sup- ported (however, SeSu has been remade and can be found in the 'trunk').

2.2.1 Work-in-progress prototypes

There are several prototypes contained within the aforementioned 'working' folder. These prototypes provide further examples of how to use the framework but remain in-complete and as such, may contain flaws and/or not fully function:

- **aMuSeV4:** an application based around children retrieving image results and using these to create stories in a comic book style format. This application is, currently, very incomplete.
- **FiFi:** this folder is a placeholder for an application deployed on a server at Glasgow - http://pooley.dcs.gla.ac.uk:8080/fifi/
- **LSee:** an application allowing children to search for a location and, from this location, retrieve a mash-up of search results (image, video, tweets and news) taken from that location. LSee (Location Search) is, functionality wise, fairly well developed but the layout and styling is very basic.
- **YouSee:** YouSee is a web application designed to provide a fun, safe, environment for children to browse videos. Videos are presented in the form of carousels. Each carousel represents a category and contains a series of videos related to it. A child using YouSee can, watch a video and browse videos in the current carousel or change to a different one. Carousels are created for the children by their parent/guardian. This application is functionally almost complete but, interaction wise, the carousel browsing is in-complete - existing only in a temporary static form.

N.B. Once completed, these prototypes will be moved to 'trunk/prototypes'.

2.3 Tag

This folder contains archived versions of the Hospital demonstrator (EmSe/Emma Search), the framework and the teaching applications (found in branches). These will only be of interest with respect to the evolution of the various parts and/or in the event of having to revert to an older version - for whatever reason.

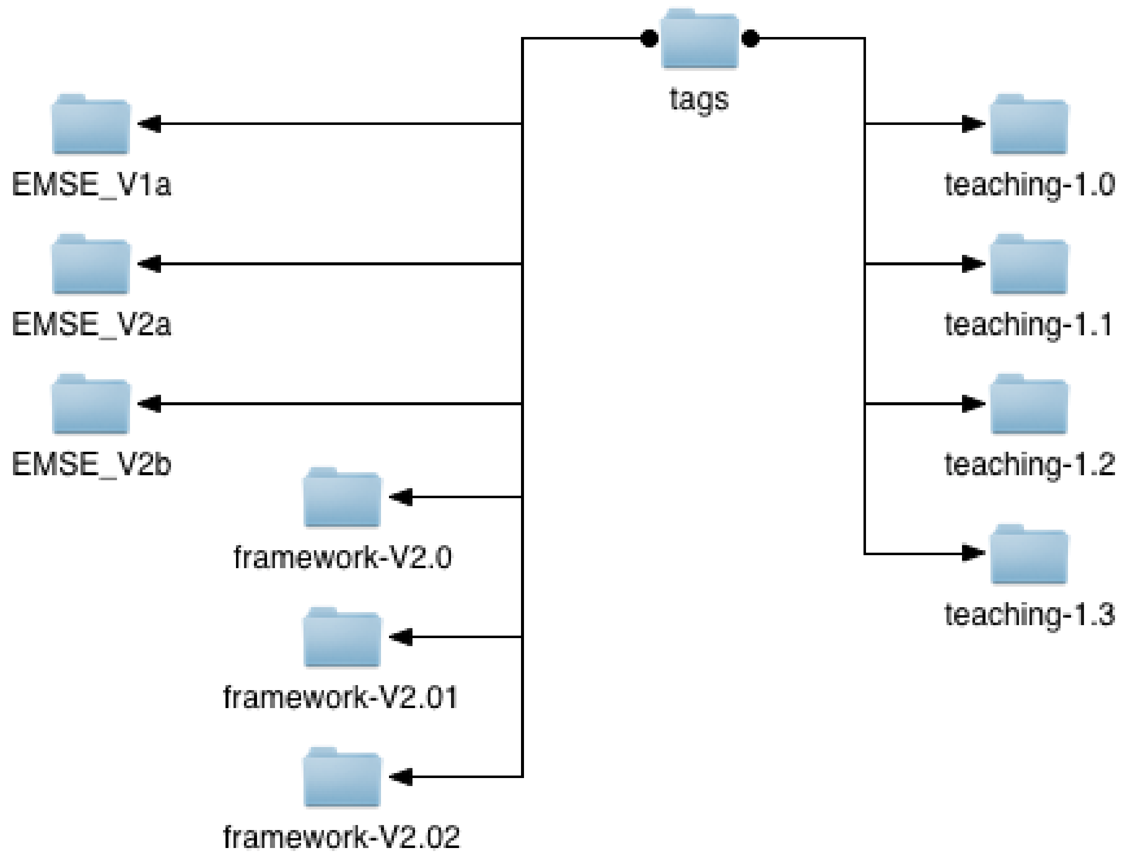

Figure 6 - *Diagram showing the structure of the 'tags' folder in the repository.*

3 About the Framework

3.1 Overview and background

The framework is part of the PuppyIR[1] project funded by the European Union, which is investigating children's information retrieval (IR). The project's long-term goal is to work towards universal access of information for both children and adults. As part of this, the framework is being developed as a suite of tools to assist developers and researchers in rapidly developing interaction IR applications for children.

In summary, it aims to:

- Simplify the process of building interactive IR services;
- provide a disparate and extensive suite of components, specifically tailored for children;
- incorporate current research findings in children's IR;
- be highly extensible (in all the main sections, and their respective components, of the framework), so that the framework can be adapted for an applications specific needs;
- and, to provide extensive documentation [this document] with tutorials detailing how to use, extend and customise all the different parts of the framework.

3.2 The main features of the framework

In this Chapter, the key functionalities of the framework are introduced and discussed; links are provided to the Chapters that provide more detailed commentary (and examples) of each functionality discussed here. To accomplish the aims listed above, the framework offers a developer, or researcher, a large variety of functionalities and associated components. These are split up into several distinct sections of the framework.

3.2.1 Data Formatting

PuppyIR provides a standardised format for both queries and the results of a search, called a response. This is so that all the components are able to interoperate and also because having this consistency, makes it easier for developers/researchers to make use of these elements in their applications. This standardised format is an implementation of the OpenSearch Standard and the frameworks model of them can be found in PuppyIR's 'query' and 'response' classes; which are used by all the components that deal with such data. Many search services and API's support this standard, but, in some cases, some processing is required - in order to present data in a form that it is compliant with the OpenSearch standard (there are many examples of this processing in the framework's search engine wrappers).

[1] For more details about the PuppyIR project, please visit the project's website at: http://www.puppyir.eu/

3.2.2 Architectural Paradigms

There are two paradigms, included with the framework, for developers/researchers to use to build PuppyIR based applications, these are:

- **One Pipeline, One Search Engine (Search Service):** this is the standard (in terms of prototype and demonstrator adoption) paradigm for creating PuppyIR based applications. In it, a unique query and result pipeline is created for each search service. A search service is then linked to a source, i.e. a search engine wrapper like Bing or YouTube so that it can retrieve and process results.
- **One Pipeline, Many Search Engines (Pipeline Service):** an alternative to the search service paradigm, where only one query and result pipeline is created, various search engine wrappers can be associated with the pipeline (defined by the pipeline service). Either 'search all', or 'search specific' (i.e. search a specific search engine wrapper associated with the pipeline service) can be used to retrieve results using the defined query & result pipelines.

A developer/researcher can select the paradigm that is most suited to their application; no matter which one is used, the same components and options (for configuring them) are available. This is due to all the components being generalised, in terms of their: interface, methods and parameters. All of the paradigms, however, make use of the 'query' and 'response' formats as mentioned earlier (however, the pipeline service returns 'response' objects in a slightly different way).

3.2.3 Event and Query Logging

Included with the framework are two kinds of logger, both of these are designed to assist developers and researchers in evaluating their applications, they are: (1) a query logger and (2), an event logger. Between these two kinds of logger, any kind of data required to be logged can be, for evaluation/analytical purposes.

Both the Search and Pipeline Services provide the ability to log queries, sent to the service in question, by a user. It is possible to log such queries at two distinct stages:

- **Un-processed:** the query passed to the service in question before it goes through the query pipeline.
- **Processed:** the query after it has gone through the pipeline (assuming it was not rejected during processing), for example it may have been extended via new terms being appended or spelling mistakes automatically being corrected.

This allows two key areas to be investigated: (1) what sort of queries the users are sending and (2), the results of the query pipeline(s), defined in the application, on these queries.

The event logger provides a developer with a component that allows them to log only the details they wish (for the event being logged) to be logged for their specific application. This is possible via a keyword arguments parameter to the log method. However, an 'identifier' and 'type' must be supplied in order to differentiate the different events and assist with categorisation for analysis of the log file(s).

3.2.4 Query and Result Filters

Filters in both the Query and Result pipelines are components which decide whether, or not, to accept a specific result (in a response) or a query - depending upon which pipeline it belongs to.

This type of filters is used mainly for avoiding profanity, both in the queries and responses. There are examples of profanity filters using both a blacklist of words or accessing to external services (like Google's one).

For responses, there are example filters that does not accept a result too complicate for a child's age, or delete results from some web sites not suitable for children.

3.2.5 Query and Result Modifiers

Filters in both the Query and Result pipelines are components which modify the query or the response. The difference between a filter and a modifier is that a filter reject (or not) the element (query or result), and the modifier always accept the element, but it can modify it.

For instance, in the case of bad words, the filter rejects a result if contains a bad word. A modifier can substitute the offending bad word with asterisk.

A modifier can also add keywords to a query in order to obtain more child-oriented results, like related terms.

3.2.6 Search Engine Wrappers

The PuppyIR framework contains a number of varied search engine wrappers. In this section, an overview of these wrappers, in terms of their category, is provided in order to provide an easy access guide to what is available (to enable a developer to select what best suits the application they have in mind).

Please note that wrappers can, and do, appear in multiple categories as some wrappers are more general purpose than other, more specific, services. Also, the generic 'web' results category is not listed but is provided by, for example, *'Bing'* please see the API guide for more.

Some of these wrappers require API keys (again, see the API reference for details) in these cases, this requires the developer to sign up for said key on the respective search service webpages for the API in question.

3.2.6.1 Book Services

- 'GoogleBooks' this wrapper provides access to the Google Books data store, you can search for books and, in some cases, retrieve samples or whole books for reading (you need to embed the samples if used in an application).

3.2.6.2 Image Services

These wrappers provide the ability to search for and retrieve image results:

- 'Bing' and 'BingV2'
- 'Flickr'
- 'Picasa'

3.2.6.3 Information Services

- 'Wikipedia' and 'Simple Wikipedia' allow the searching of wikipedia's database. They results consist only of a link, snippet (summary) and a title, hence, this is not suitable if you require a large amount of textual content; but ideal for providing a short description for children.

3.2.6.4 Location Based Searching

These services allow for searching for: either results in a defined location or for a location itself.

N.B. Google Geocode should be used to retrieve the geo-coordinates and/or bounding box to use with the other services as their location based parameter(s).

For an example of this in action, a prototype (it should be noted, that this prototype was abandoned and the code is quite rough in addition to it not being styled) is available which you can download via:

```
$ svn co
https://puppyir.svn.sourceforge.net/svnroot/puppyir/branches/working/
LSee LSee
$ cd LSee
$ python manage.py runserver
```

Visit: http://localhost:8000/lsee

- 'Flickr' allows for the retrieval of geotagged images within a defined bounding box.
- 'Google Geocode' this service allows results to be retrieved for locations, for example, if you search for 'Edin- burgh' it will return details of the various Edinburgh's around the world (like their location/latitude).
- 'Twitter' allows for the retrieval of geotagged tweets made within a box defined by a point - the origin - with a radius to define a box around the point.
- 'YouTubeV2' allows for the retrieval of geotagged videos from within a box defined by a point - the origin - with a radius to define a box around the point.

3.2.6.5 Movie Services

- 'Rotten Tomatoes' allows for the retrieval of details about movies like: the cast and aggregated review score etc.

3.2.6.6 Music Services

Note: YouTube and YouTubeV2 are also, arguably, a music based services due to the large proportion of music based content.

- 'iTunes' The iTunes wrapper allows you to search for not only music, but also forms of media such as movies and TV shows.
- 'LastFM' Allows for searching for music using LastFM, specifically, you can search for: tracks, albums and artists.
- 'Soundcloud' This wrapper allows you to search for music on the Soundcloud service, using advanced searching parameters like genre and beats per minute.
- 'Spotify' Allows for searching for music using Spotify (and get links to play the songs), specifically, you can search for: tracks, albums and artists.

3.2.6.7 News Services

These wrappers provide the ability to search for news stories

- 'Bing' and 'BingV2' allow for the searching of the 'news' results.
- 'Guardian' is a wrapper for the search API of the UK based newspaper: the Guardian. While UK based this service also provides a large variety of stories about events the world over.

3.2.6.8 Social Network and Social News Services

- 'Digg' a social news website for sharing items which are then rated by the community; this acting as a method of filtering the quality of results.
- 'Twitter' a social network for posting short messages.

3.2.6.9 Spelling Suggestions and Dictionary based results

- ‘BingV2’ using the ‘spell’ source type provides spelling corrections to a query.
- ‘Wordnik’ this service provides a spelling correction feature, in addition to providing definitions of words and examples of words in context (via selections of text from various web pages).
- ‘WebSpellChecker’ allows for searching for spelling corrections in a variety of languages - there is no extra information returned however just the spelling correction suggestion.

3.2.6.10 Video Services

These wrappers provide the ability to search for videos. It should be noted that Bing’s search engine strongly favours YouTube results so there is a lot of overlap if both it and YouTube are used in the same application.

- ‘BingV2’, described above, can return video contents.
- ‘YouTube’ and ‘YouTubeV2’ results from YouTube come with an embed URL so they can be played in-line (as seen in the MaSe and aMuSeV3 prototypes).

3.2.6.11 Auxiliary services

These query engines does not search in the public search engines available in the Internet.

- ‘Echosearch’: This is a false query engine for debugging applications.
- ‘Enmasearch’: This query engine is used for searching the database in the Enma Hospital.
- ‘Opensearch’: This example query engine can be easily adapted for any public search engine compliant with OpenSearch standard.
- ‘Solr’: Used for searching in a local engine constructed over Solr.
- ‘Woosh’: Search service for query log data held in a Whoosh query index

3.3 Extensibility of the framework

3.3.1 Search Engine Wrappers

It is expected that the area, of the framework, especially, is one with great potential for future expansion and development. This is due to the inevitable influx of new API’s and updates to the ones currently supported by the PuppyIR framework. The section detailing this area (in the Technical Annex), therefore, looks at how to write new wrappers that are compatibly both with the architectural paradigms as well as the other components that interact with search engine wrappers (i.e. filters etc).

3.3.2 Query and Result Filters &Query and Result Modifiers

The other areas identified as being a likely candidate for extension, are the filters and modifiers available in both the query (see section *6.4.1 Extending the Query Pipeline*) & result (*6.4.2 Extending the Result Pipeline*) pipelines.

A lot of the filters and modifiers included with the framework were developed as part of, or in response to, the latest research in children’s information retrieval hence, this being a likely area to get added to as researchers/developers explore new methods & techniques.

3.4 Paradigm 1: One Pipeline, One Search Engine

The core component of a PuppyIR based application is a search service in this paradigm. A search service contains a variety of individual components that, when combined together, allow for: searching, retrieving and processing the results - from a specific defined search engine. These search services are stored and managed by a service manager. The diagram below shows the structure of a search service from its owner, the service manager, to all the individual components contained within the search service.

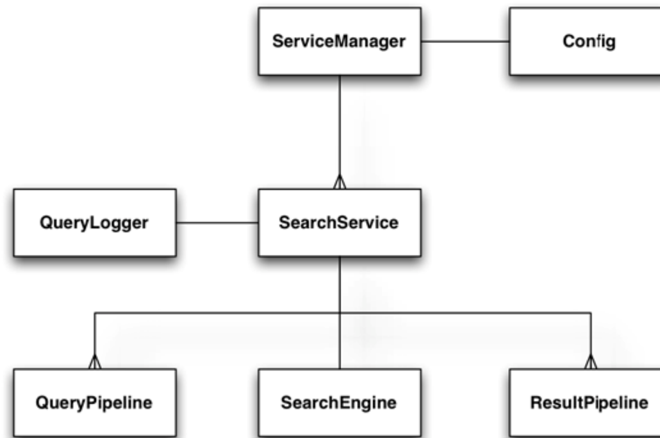

Figure 7 - *The basic architecture of a PuppyIR application, using the 'Search Service' paradigm.*

3.4.1 Description of the components

The roles of the components are as follows:

- **Service Manager:** this is in charge of managing (adding and deleting) all the search services used by an application.
- **Config:** local configuration options (e.g. for proxies, API keys and log files).
- **Search Service:** a single search service, with its own query logger and distinct query & result pipelines.
- **Query Logger:** logs queries, sent to a search service, to file (available for both un-processed and processed query logging - more on this later).
- **Search Engine:** this is the search engine wrapper for a specific 'search service' - e.g. a 'search service' that uses the YouTube search engine (wrapper).
- **Query Pipeline:** a collection of query filters and modifiers associated with a specific 'search service'.
- **Result Pipeline:** a collection of result filters and modifiers associated with a specific 'search service'.

3.4.2 Data flow in the 'Service' paradigm

The diagram below shows the basic flow between a user issuing a query and their results being returned.

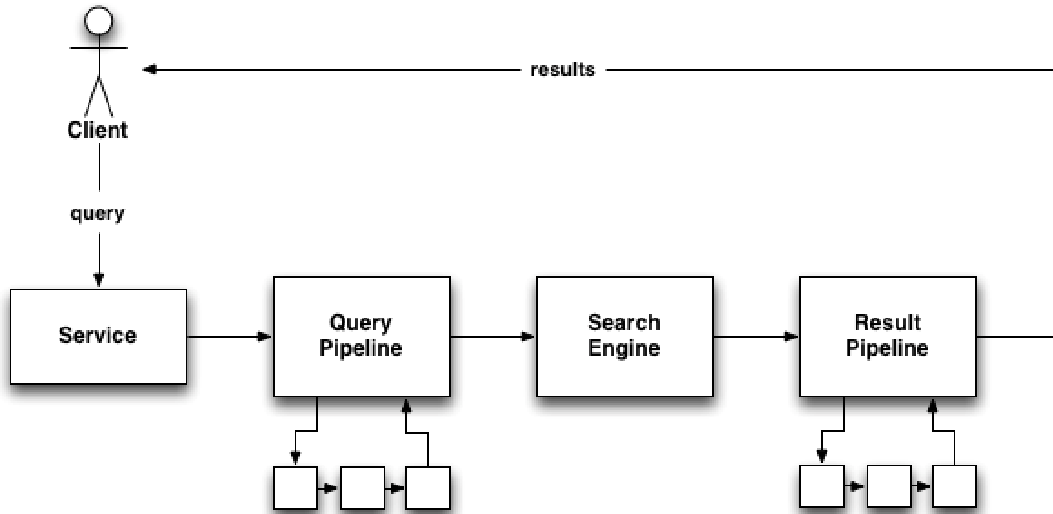

Figure 8 - *The basic data-flow in the 'Search Service' paradigm.*

The 'search service' is passed a query, by the user/client, via the search method in the 'search service' (simple search is also available; this skips the query and result pipelines). It then goes through the query pipeline, first running all the query filters and then all the query modifiers. The processed query is then passed to the 'search engine' (defined for the current 'search service') and the results retrieved using the search method contained in the 'search engine' wrapper.

The results are then passed through the result pipeline, first by running all the result filters and then, finally, all the result modifiers. Following the completion of the 'result pipeline', the processed results are then returned to the user/client.

3.4.3 The Query and Results formats

Referring to the data flow diagram above, the formats of a query and results are as follows:

- A query is in the 'Query' format.
- The results are in the 'Response' format; this is what is returned by the search call (for the search engine in question).

Both the Query and Response formats are implementations of the OpenSearch specification

3.5 Paradigm 2: One Pipeline, Many Search Engines

The core idea behind this alternate paradigm is that you create and manage one pipeline - to which search engines can then be added. This is in contrast to the 'search service' paradigm, where each search service, and its associated search engine wrapper, has its own distinct pipeline. Like with the 'Search Service' paradigm, there is a query pipeline and the result pipeline, but, in addition to this, there is an additional pipeline: the search engine pipeline (which makes use of a search engine manager; this is equivalent, in most respects, to the 'Search Service Manager' from the 'search service' paradigm).

The picture below shows how all these components relate to each other:

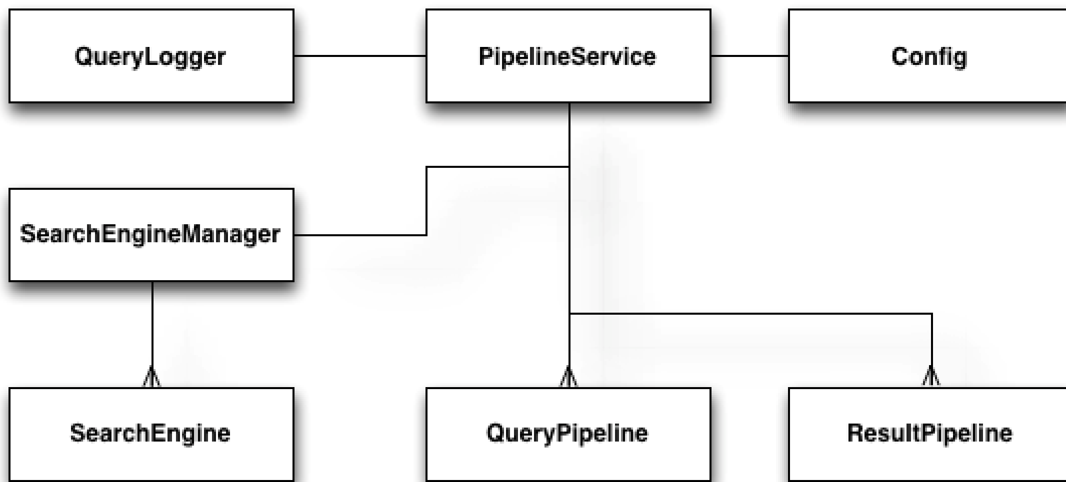

Figure 9 - *The basic architecture of a PuppyIR application, using the 'Pipeline Service' paradigm.*

3.5.1 Description of the components

Each of the key components, shown in the picture above, are summarised (in terms of how they relate to this paradigm) below; except for the 'Query Logger' and 'Config' components as these are identical to those found in the 'Search Service' architecture.

The roles of the components are as follows:

- **Pipeline Service:** this is the main component in this paradigm as it is in charge of managing and running the pipeline it defines (i.e. all the filters and modifiers). It also contains the next key component, the 'search engine manager'.
- **Search Engine Manager:** this component is, roughly, equivalent to the 'search service manager' as found in the 'search service' paradigm; except that it manages search engines as opposed to search services. Its main tasks are adding and removing search engines.
- **Search Engine:** this is the component managed by the search engine manager and is the same as in the 'search service' paradigm; except that it's linked to the 'pipeline service' not a search service. Like in 'search service' each search engine has a name assigned to it and the 'search engine manager' looks for, deletes and retrieves search engines using this variable.
- **Query and Result Pipelines:** these are exactly the same as their counterparts in the 'search service' paradigm, excepting that they are stored by a 'pipeline service'.

3.5.2 Data flow in the 'Pipeline' paradigm

The data flow in this paradigm is a little more complicated than in the 'search service' paradigm, due to the extra complexity introduced by having multiple search engines associated with one pipeline. The picture below shows the data flow between a user issuing a query and their receiving the result(s) of this query.

The 'pipeline service' is passed a query, by the user/client, via one of two methods: search all or search specific. From here, the query pipeline is run (once; even if there are multiple search engines - since they all have the same query and query pipeline), first going through all the filters and then all the modifiers. Following this, the 'search engine manager' is called to retrieve either: all the search engines it manages, or one specific one. The next step is to run through the 'search engine pipeline' with the results of the previous step. (1) shows the entry point for this process, at this stage either each search engine will be processed in turn

or, in the case of a specific search, only one will be processed (as defined by the search specific call).

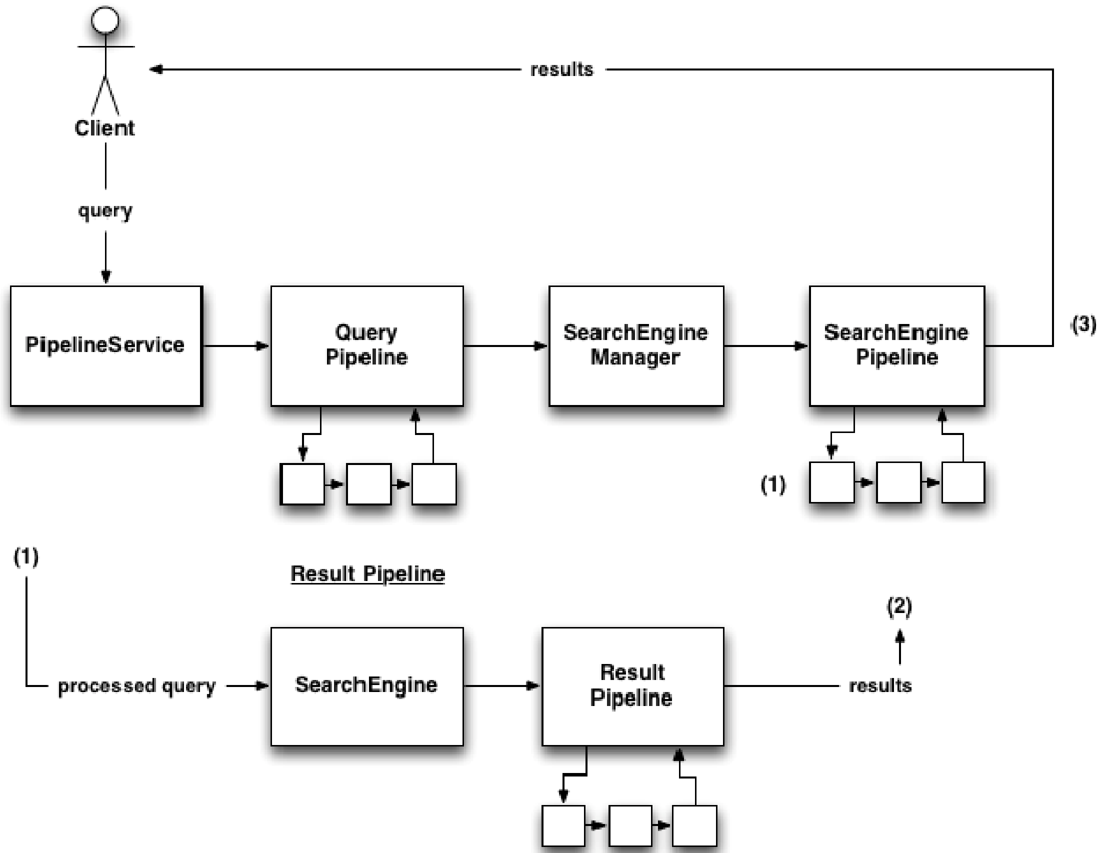

Figure 10 - *The basic data-flow diagram for the 'Pipeline Service' paradigm.*

In the above diagram, the section under the label 'Result Pipeline' shows how the processing of a search engine works:

- the processed query is passed to the current search engine (going through the pipeline);
- next, the search method for this search engine is called and the results retrieved;
- then the result filters, followed by the result modifiers are run (this step is the same as the result pipeline from 'search service' - just applied to each search engine in turn);
- lastly, the results from the current search engine are added to the overall 'results' at (2).

Once the above process has been completed, for each search engine, the overall 'results' are returned - (3) shows the point at which the overall 'results' are complete and can then be returned to the user/client.

3.5.3 The Query and Results formats

Referring to the data flow diagram above, the formats of a query and results are:

- A query is in the 'Query' format.
- The results format is a Python dictionary, with one entry for each search engine; the key being the named assigned to the search engine and the value being the response object returned from the search call (for the search engine in question).

Both the Query and Response formats are implementations of the OpenSearch specification.

3.5.4 Possible advantages of using this architecture

This paradigm has the potential to be more efficient than the ‘search service’ paradigm, in terms of code and effort on the part of a developer/researcher, in the following ways:

- If you want the same pipeline (filters etc) for multiple services you only need to set the pipeline up once and can just add the search engines you want to the ‘search engine manager’ (contained by your ‘pipeline service’).
- Related to the above point, is that the Query pipeline is only run once with ‘searchAll’ because all the search engines use the same pipeline.
- Less code for getting results - just a simple ‘searchAll’ call rather than a search call for each search service and the associated code to handle this.

4 Development Roadmap

The core components of the framework have stabilised, and will be refined further for the final release. Deliverable [8] D4.4 – Release of Open Source Framework V1.0 highlighted known issues and future developments which form the basis of the roadmap for the remaining development of the framework and additional components.

The first issue of documentation of the framework has been addressed, as discussed in section 3 of this deliverable. The second issue of client interface development is being addressed as part of work package 7, through the development of two web based interfaces: the hospital information service (see D7.3) and the post-visit Museon information service (see D7.2), both of which are built using the framework.

The remaining issues that are under development are integration and configuration of PuppyIR services. Integration refers to the link between the framework and the Django web application framework. Configuration refers to the means by which a service can be altered to fit the requirements of its users. Each issue is discussed briefly in the following sections.

4.1 Integration of PuppyIR and Django

An early objective of the framework design was to keep it modular and independent of any third party web application framework. This has the benefit of allowing the framework to be explored in a standalone mode. This removes the burden of needing to also become familiar with a web application framework. Furthermore, the framework documentation has a section devoted to showing framework users how to build a complete search service that simply runs on the command line.

However, it is most probable that users of the framework will want to go beyond the command line to create an interesting service for children. This entails the development of a client interface and the hosting of a service so that it is web accessible. Throughout the project, developers have relied upon the Django web application framework to support these requirements. As this has been a successful combination in terms of *stability* (Django is a mature and well supported open source project) and *efficiency* (Django drastically speeds up the development of web applications), it has become the recommended method for third party developers to adopt.

Therefore, to reflect the integration of PuppyIR and Django a suite of basic setup utilities/scripts and documentation will be developed that achieve the following aims:

1. Create a basic Django web-application
2. Modify this web-application to become a PuppyIR service, providing a:
    a. Skeleton PuppyIR service project layout
    b. Default search service engine
    c. Default search service interface
    d. Default admin interface to configure search service

Ideally, this process would be encapsulated in several simple commands that a developer is required to execute. Afterwards, the developer should be able to visit a web page and use the default service without having written any code. It is foreseeable that this process could be repeated for any compatible web application framework, should a better alternative emerge.

4.2 Configuration of PuppyIR Services

Once a service is up and running, it may require configuration. The most desirable method of achieving this is through a web-based administrative interface. A further advantage of Django is the provision of a comprehensive web-based admin interface. For the purposes of PuppyIR, it would be possible to use this admin interface to access a model of the service, manipulate it, and save new configurations, ideally without having to restart a service.

To illustrate this, a PuppyIR service can be modelled as the following set of relationships:

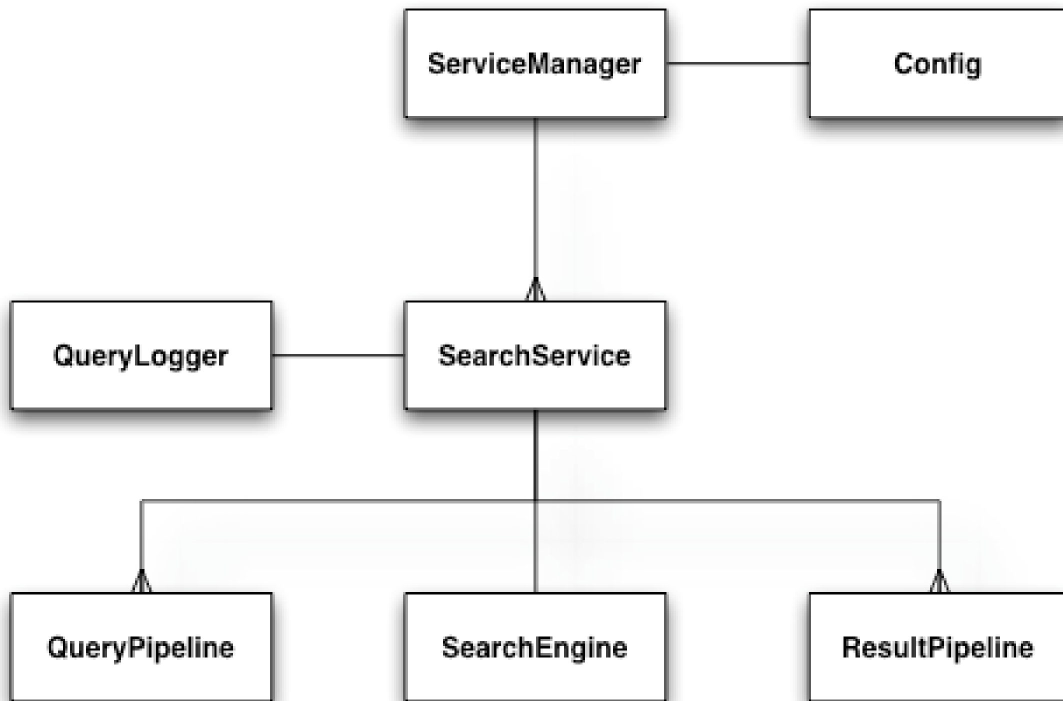

Figure 11: *Abstract representation of a PuppyIR service model*

The idea is that this data model of a service could be modelled as a 'model' for Django, and stored in a database. The Django admin interface could then access and manipulate the Django model. The PuppyIR service could also be created using this model, and whenever it changes, it should reconfigure itself.

Currently, an early version of this admin interface for configuration exists. The main points of future improvement are:

- Currently, if the model is changed by the Django admin interface, a running PuppyIR service does not realize about it and the service does not reconfigure itself. Monitoring a change in the model is not easy, because the 'listeners' provided by Django are unsuitable for this case.
- Ideally, the configuration function should be encapsulated a completely independent library. Unfortunately, in the case of an application using Django models for other tasks, the configuration model and the application model must integrated the general model of the application. Newer versions of Django provide better support for several models in an application, so this issue can be improved in the future.

A tutorial about using the current implementation can be found in the Annex, section 6.3.5 in this document.

5 Summary

This report outlines version 3.0 of the framework, showing the main trunk of development and the various branches that have emerged throughout the progress of the project; the current set of features that are included in the core framework release; a brief overview of the public documentation and its contents; the dependencies and installation requirements to get started with the framework (in the annexes); and the main developments that have been incorporated into the third and final release within the project lifecycle.

6 Annexes

6.1 Annex A: Installing the Framework

6.1.1 Requirements and Installation

The PuppyIR framework is Python-based and requires, in addition to Python itself, several external dependencies. It can either be installed as a standalone service, or combined with the Django web application framework to build web services. The requirements, both basic and those required for additional functionality, are detailed here.

Note: if you are running MacOS X, please ensure that you have X-Code installed (either Version 3 or 4; this may be included on your install disc). This is required as several of the dependencies use X-Code's C compiler.

6.1.1.1 PuppyIR and MacPorts

For developers using MacOS X, MacPorts can be used to install all the PuppyIR framework requirements. If you wish to install these using MacPorts please ensure that you install the *'py27'* versions. Please consult the MacPorts documentation for how to use MacPorts and then install all the basic and extra requirements (if required) using their port versions.

The one exception to the naming convention (of *'py27')* is setuptools, which has the Port name 'py-setuptools'.

6.1.1.2 Basic Requirements

The basic PuppyIR framework installation requires all of the following to be installed (in addition to the framework itself):

- Python Programming Environment (N.B. Python 3.x is not supported)
- Setuptools
- Universal Feed Parser
- lxml
- BeautifulSoup

6.1.1.3 Extra Requirements

The following external dependencies are only required, if you intend to do any of the development tasks detailed below.

To create web services using the Django framework and/or to run the various prototypes and demonstrators:

- Django Web Application Framework

To run some of the prototype services included with the PuppyIR framework (specifically the JuSe prototype):

- PIL (Python Imaging Library)

If you require the use of the 'spelling modifier' install Enchant:

- Enchant

If you require the use of a full text indexer:

- Whoosh

If you wish to use the 'SuitabilityFilter[2]' to filter results and/or make use of Strathclyde University's work in 'trunk/interfaces') you will need to install Java.

6.1.1.4 Basic Installation

The following sections provide instructions on installing each of the requirements.

6.1.1.5 Install Python

If your system does not have Python installed, or you have an earlier version, you can find the latest 2.7 branch of Python here. Follow the installation instructions for your own operating system.

At present, Python 3.x is not supported and may cause problems if installed. You can discover your current version of Python by launching a command prompt and typing the command 'python'. The version number should be displayed as shown below. If Python 3.0+ is installed, please install the earlier version (the 2.7 branch) to run PuppyIR.

```
$ python
Python 2.7.1 (r271:86882M, Nov 30 2010, 10:35:34)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type 'help', 'copyright', 'credits' or 'license' for more
information.
>>>
```

6.1.1.6 Install Setuptools

This is a pre-requisite to allow several of the other basic dependencies to be installed. Download the source from http://pypi.python.org/pypi/setuptools

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

6.1.1.7 Install Universal Feed Parser

This allows PuppyIR to parse RSS and Atom feeds. Download the source from http://code.google.com/p/feedparser/

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

[2] For the 'SuitabilityFilter' to work you need to have java added to your system path; how to go about this varies depending on the Operating System (OS) you are using - there are many articles on the internet explaining how to do this for all the major OS's so this is not detailed here.

6.1.1.8 Install lxml

This allows PuppyIR to parse XML files. Download the source from http://pypi.python.org/pypi/lxml/

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

6.1.1.9 Install BeautifulSoup

This is a HTML/XML parser, one of its main functions is handling tree traversal automatically. Download the source from http://www.crummy.com/software/BeautifulSoup/#Download

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

6.1.1.10 Installing the PuppyIR Framework

There are two options for installing the PuppyIR framework itself, either you can install the latest development version, or, install a specific release of the framework.

Option 1: Installing a specific release

If you require a specific release, for your application, or simply wish to use a release that is likely to be stable, then choose this option and follow the instructions below.

Download the specific release you want from http://sourceforge.net/projects/puppyir/files/release/ then:

```
$ cd path/to/puppyir
$ python setup.py install # may require 'sudo'
```

Option 2: Installing the development version

Alternatively, the very latest release - the development version - can be checked out of the repository by following the instructions below.

N.B. the development version is not guaranteed to be stable and may be incompatible with certain prototypes and/or demonstrators.

6.1.1.11 Installing the Extras

The following sections provide instructions on installing each of the extra requirements.

6.1.1.11.1 Install Django

(Only required if building web services that require or make use of the Django web application framework)

Django is a Python based web framework designed to build web applications quickly, installing this allows develop- ers/researchers to take advantage of the many features offered by Django and also to run the prototypes and demon- strators bundled with the framework.

Download source from https://www.djangoproject.com/download/

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

6.1.1.11.2 Install Python Imaging Library (PIL)

(Only required for the JuSe prototype)

This is a library that provides allows various image processing tasks to be done on a large variety of image formats. Download the source from http://www.pythonware.com/products/pil/

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

6.1.1.11.3 Install Enchant

(Only required if using the 'spelling modifier')

This is a library that checks the spelling of words and provides a list of suggested correct spellings.

It requires enchant library (version 1.5.0 or greater) which can be downloaded at http://www.abisource.com/projects/enchant/ - installation instructions can be found on this site as well.

Then download Enchant for Python from http://packages.python.org/pyenchant/

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

6.1.1.11.4 Install Whoosh

(Only required for local full text indexing and to run certain prototypes & demonstrators) Download source from http://pypi.python.org/pypi/Whoosh/#downloads

```
$ cd /path/to/source
$ python setup.py install # may require 'sudo'
```

6.1.2 Other features and framework support

6.1.2.1 Which version of Python is the framework for?

The PuppyIR framework is designed, built and maintained using Python 2.7; Python 3.x is not supported and earlier versions may have compatibility issue. It is, therefore, recommended to upgrade to Python 2.7 rather than using earlier versions.

6.1.3 Standalone Services

The PuppyIR framework can be used to build a standalone service for research and development purposes. This mode has minimal requirements and simplifies the process of building custom search services that do not require a user interface.

6.1.4 Proxy Server Support

Many workplaces and research institutions use a proxy server and so, any applications created, using PuppyIR, would need to go through such a proxy server. The framework, therefore, offers a simple interface for its components that enables developers/researchers to easily set-up the components they are using to work with a defined proxy server. The code below shows how to create a service in both the paradigms, included with the framework:

```
# Set-up a config setting for a proxy server
```

```
config = {'proxyhost': 'http://your-proxy-server-address'} # --
Paradigm 1 and proxy servers --

# ----------------------------------
# Create a service manager and set it to use config
sm = ServiceManager(config) # Create a search service for Bing Web

ss = SearchService(sm, 'bing_web') # Set our new search service to
use the Bing wrapper

ss.search_engine = Bing(ss) # Add new search service to
ServiceManager

sm.add_search_service(ss) # -- Paradigm 2 and proxy servers --

# ----------------------------------
# Create a Pipeline Service called 'myPipeline' using config
pipelineService = PipelineService(config, 'myPipeline') # Create a
Bing search engine wrapper

bing = Bing(pipelineService)
# Add Bing to our search engine manager (this stores all our search
engines)
pipelineService.searchEngineManager.add_search_engine('Bing', bing)
```

6.1.4.1 Django support

The PuppyIR framework can be integrated with the Django web application framework to provide a toolkit for rapidly prototyping and deploying search services for children on the web. PuppyIR includes a number of components that augment the existing Django functionality.

N.B. Django is provided as an example, the framework can also work with other Python based web application frameworks as no parts of the framework are tied into Django.

6.2 Annex B: Using the Framework

6.2.1 Prototypes

This section provides examples of some of the prototypes included with – and build using – the PuppyIR framework.

6.2.1.1 Running Prototypes

Several prototype services are available as examples of how children's information services can be built using the framework.

- **aMuSeV2:** a child friendly Multimedia Search engine mash-up search service allowing YouTube, Bing Image and Picassa to be searched for videos and images.
- **aMuSeV3:** a child friendly search service allowing for the retrieving of video & image search results and the exploration of these results via the generation of new queries.
- **BaSe:** Basic Search - a bare bones search service with no frills.
- **IfSe:** Information Foraging Search, an application created as a tutorial for using the PuppyIR framework to create a pipeline.
- **MaSe:** Multimedia Search Engine Mash-up: an application created as a tutorial for using PuppyIR to create a customisable web application and in doing so, introduce web programming to school children.
- **SeSu**: Search and Suggest - a search service which filters results by their suitability for children as well as providing search suggestions for new queries.

6.2.1.2 Downloading The Prototypes

All the prototypes require Django to be installed to use them.

In addition, IfSe and MaSe also require Whoosh to be installed to use them, if you do not have Whoosh installed please visit the installation page, as detailed above, and install Whoosh.

The source code for all the prototype services can then be downloaded as follows:

```
$ svn co
https://puppyir.svn.sourceforge.net/svnroot/puppyir/trunk/prototypes
prototypes
```

To download a specific prototype, use the command as follows substituting in the name of the application (in this case, you will need to amend the paths to run the prototypes by removing the 'prototypes' part of the path as noted in the run sections of this page):

```
$ svn co
https://puppyir.svn.sourceforge.net/svnroot/puppyir/trunk/prototypes/
<APPNAME> <APPNAME>
```

6.2.1.3 Using aMuSeV2: a Multimedia Search engine mash-up

aMuSeV2 allows video and picture results to be retrieved from YouTube, Bing and Picasa. The results are then scaled according to relevance; all results are draggable allowing them to be arranged as the user wishes.

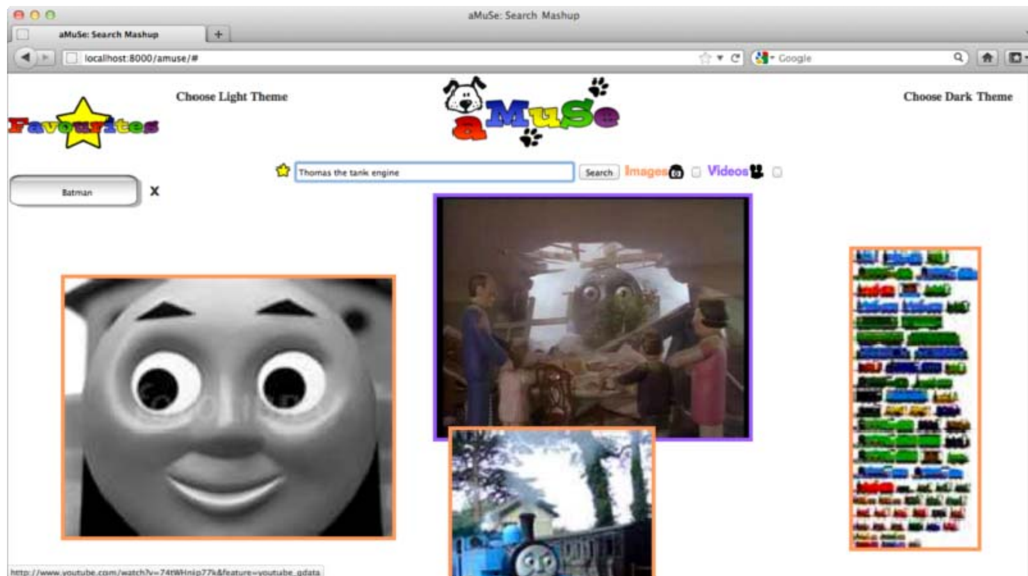

Figure 12 - *aMuSeV2 showing a video/image collage of 'Thomas the Tank Engine' results.*

Running aMuSeV2

```
$ cd /path/to/prototypes/amuseV2
$ python manage.py runserver
```

Visit: http://localhost:8000/amuse

6.2.1.4 Using aMuSeV3: a Multimedia Search engine mash-up and browsing application

aMuSeV3 allows video and picture results to be retrieved from YouTube and Bing image search. The results are then, randomly (albeit, with a left-right-top-bottom approximation of relevance), arranged in a collage of images and videos. Videos can be played in-line; clicking on an image will generate a new query which will return a new collage of results.

aMuSeV3 is only compatible with Python Version 2.7 - if you have an earlier version then please install Python 2.7 to use this prototype.

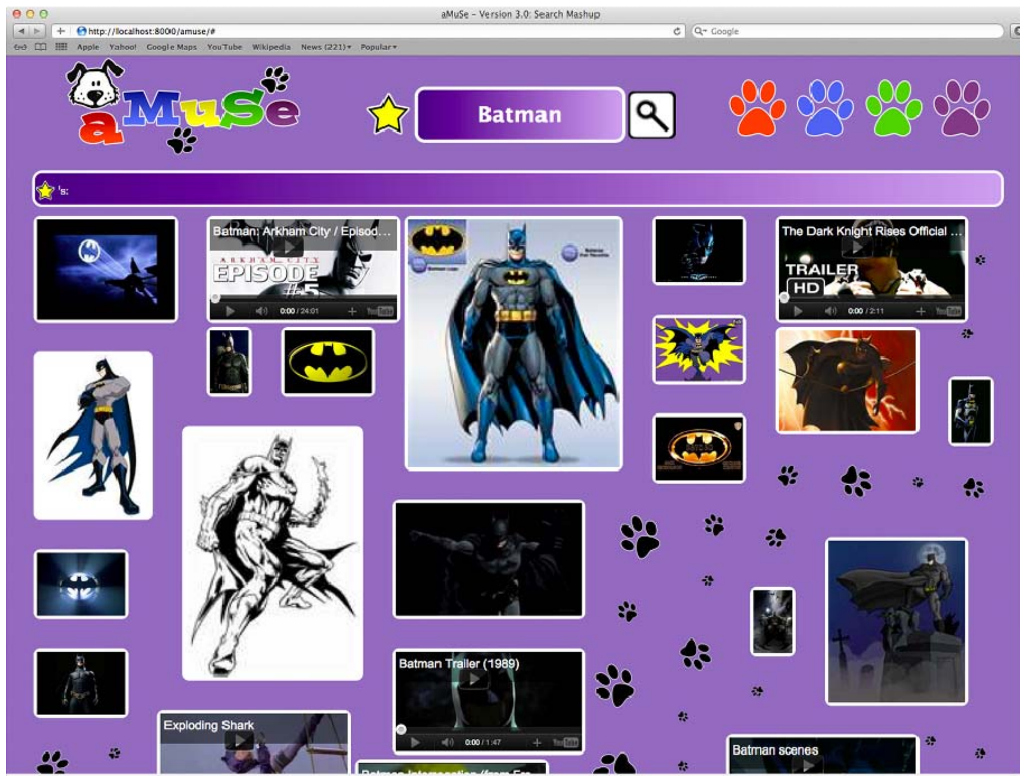

Figure 13 - *aMuSeV3 showing a video/image collage of 'Batman' results.*

Running aMuSeV3

```
cd /path/to/prototypes/amuseV3
$ python manage.py runserver
```

Visit: http://localhost:8000/amuse

6.2.1.5 Using BaSe: Basic Search

This is a simple 'google-like' interface to illustrate a simple search interface.

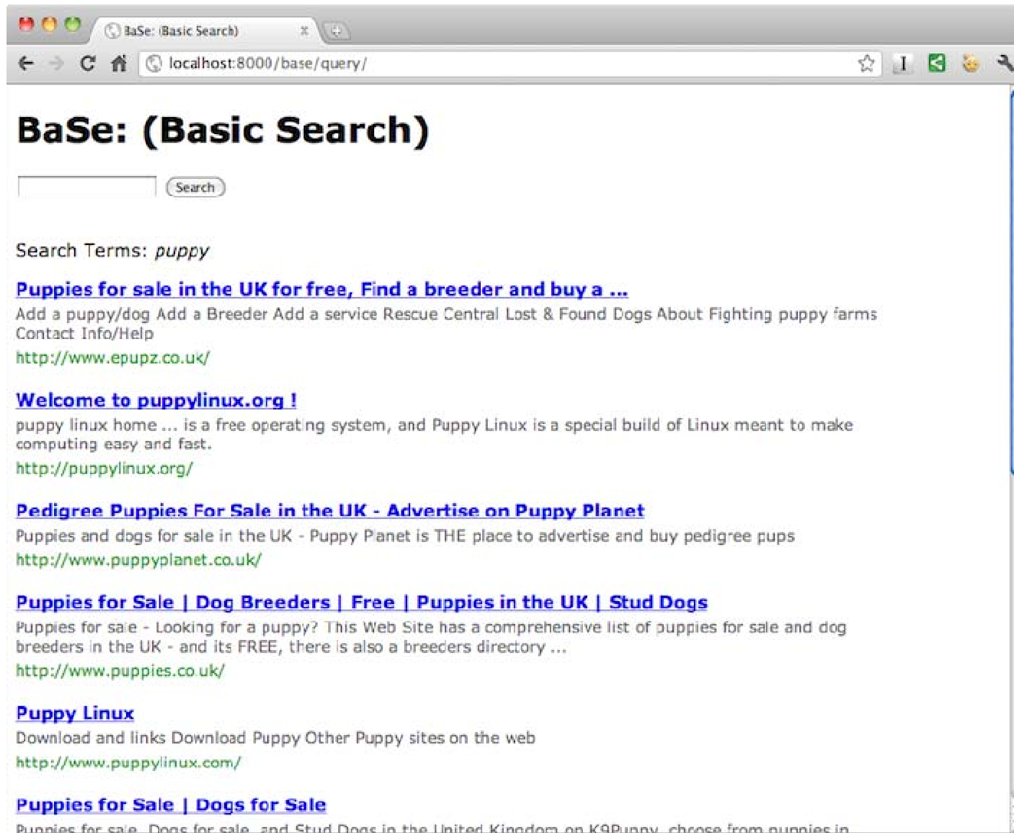

Figure 14 - *BaSe running on a local machine showing web results for the query 'puppy'.*

Running BaSe

```
$ cd /path/to/prototypes/base-tutorial
$ python manage.py runserver
```

Visit: http://localhost:8000/base

6.2.1.6 Using IfSe: Information Foraging Search

This prototype is part of a tutorial that teaches how to go about retrieving results from search engine wrappers, logging queries, generating suggestions and how to filter queries.

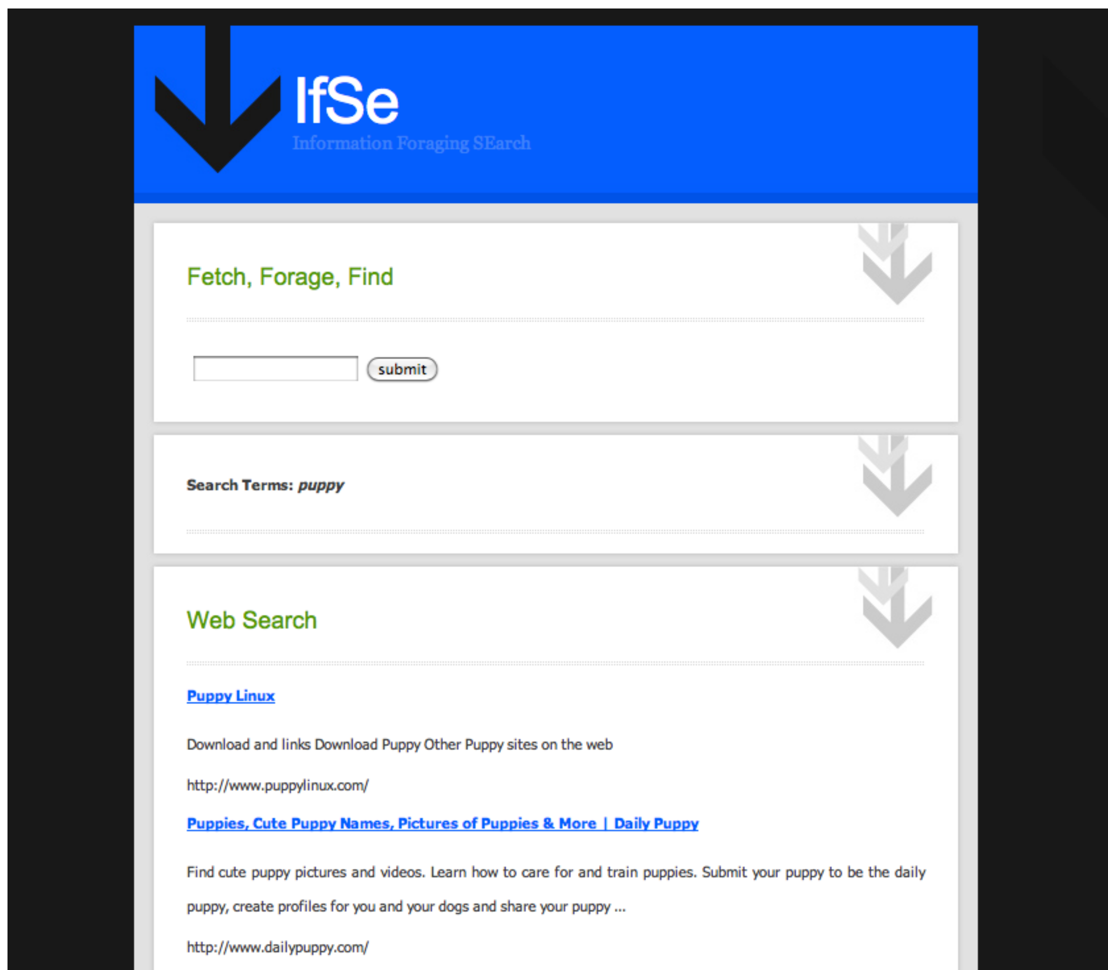

Figure 15 - *IfSe running on a local machine showing web results for the query 'puppy'.*

Running IfSe

```
$ cd /path/to/prototypes/ifse-tutorial
$ python manage.py runserver
```

Visit: http://localhost:8000/ifse

6.2.1.7 Using MaSe: Multimedia Mash-up Search Engine

MaSe is an application designed to allow children to create and customise their own search engine - retrieving results from a variety of sources and formats.

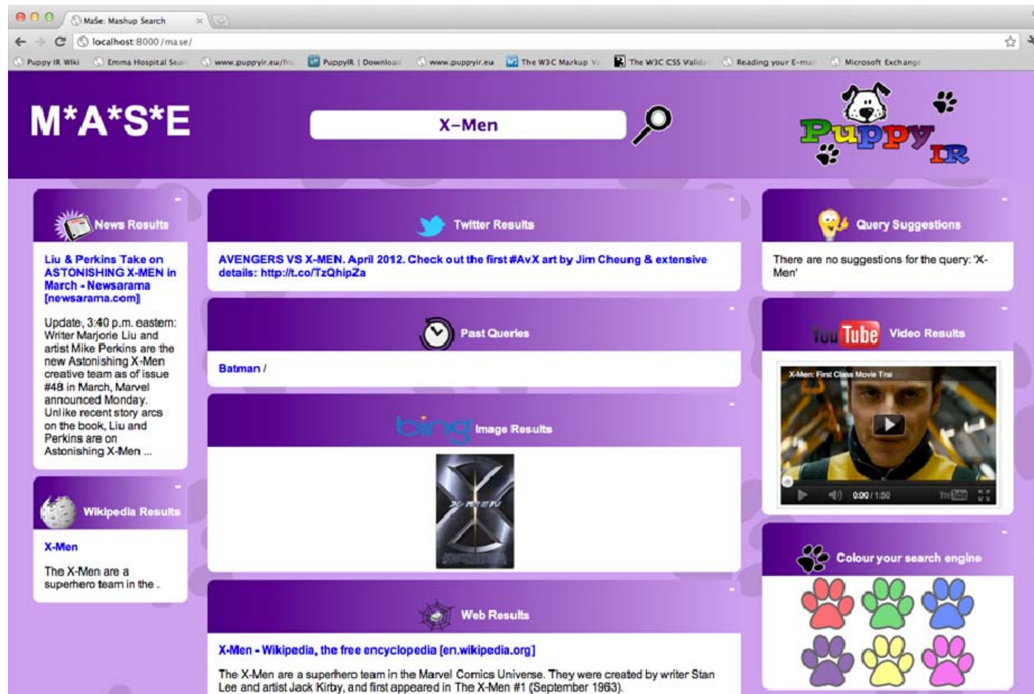

Figure 16 - *MaSe running on a local machine showing web results for the query 'X-Men'.*

Running MaSe

```
$ cd /path/to/prototypes/mase-tutorial
$ python manage.py runserver
```

Visit: http://localhost:8000/mase

6.2.1.8 Using SeSu: Search and Suggest [15]

SeSu is a prototype service that investigates query suggestions and suitability filters for children's web search.

More details can be found in [15] 'Web Search Query Assistance Functionality for Young Audiences'

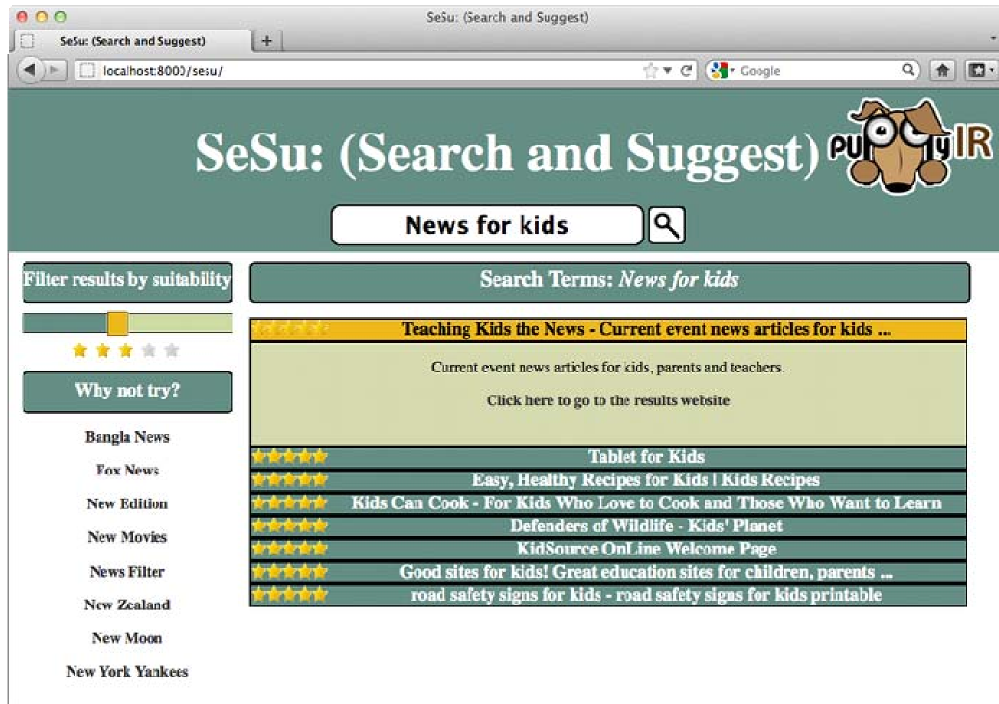

Figure 17 - *SeSu showing results, with their suitability rating, for a query about the news.*

Running SeSu

```
$ cd /path/to/prototypes/sesu
$ python manage.py runserver
```

Visit: http://localhost:8000/sesu

6.2.2 Building a Standalone PuppyIR Service

The PuppyIR framework can be used to build a standalone service with no user interface. This is a good place to start when initially developing with PuppyIR and can also be more appropriate for experimental development of new child-friendly information processing components.

6.2.2.1 Service Implementation

The following steps will create and configure a new service, consisting of: a search engine, a query logger & filtering for both the queries and the results retrieved (from the search engine wrappers used).

1. Create and Configure the ServiceManager

Create a new python script, e.g. service.py

2. Create a SearchService

```
from puppy.service import ServiceManager
config = {}
```

```
# Create the ServiceManager
sm = ServiceManager(config)
# new imports
from puppy.service import ServiceManager, SearchService config = {}
sm = ServiceManager(config)
# create SearchService and give it a name
ss = SearchService(sm, 'bing_web')
# Add new SearchServices to ServiceManager
sm.add_search_service(ss)
```

3. Add a SearchEngine

```
from puppy.service import ServiceManager, SearchService # new imports

from puppy.search.engine import Bing

config = {}
sm = ServiceManager(config)
ss = SearchService(sm, 'bing_web')
sm.add_search_service(ss)
# Assign new Bing SearchEngine to SearchService
ss.search_engine = Bing(ss)
```

4. Perform a Search

At this stage, we can now use the service we have created to search Bing.

```
from puppy.service import ServiceManager, SearchService from
puppy.search.engine import Bing

# new imports

from puppy.model import Query, Response

config = {}

sm = ServiceManager(config)
ss = SearchService(sm, 'bing_web')
sm.add_search_service(ss)
ss.search_engine = Bing(ss)
# make a new Query and search
query = Query('puppy')

results = sm.search_services['bing_web'].search(query).entries
# print results
for result in results:
      print result['title']
      print result['summary']
      print result['link'] + '\n'
```

5. Enable the QueryLogger

It may be useful to start logging queries to file.

```
from puppy.service import ServiceManager, SearchService

from puppy.search.engine import Bing
```

```
from puppy.model import Query, Response

# new imports

from puppy.logging import QueryLogger

config = {'log_dir': '/path/to/log/directory',
      # Set this to where you want the logs stored

}

sm = ServiceManager(config)
ss = SearchService(sm, 'bing_web')
sm.add_search_service(ss)
ss.search_engine = Bing(ss)
# Assign QueryLogger to SearchService
ss.query_logger = QueryLogger(ss, log_mode=0)
query = Query('puppy')
results = sm.search_services['bing_web'].search(query).entries
for result in results.entries: print result['title']
      print result['summary']
      print result['link'] + '\n'
```

6. Adding QueryFilters and ResultFilters

```
from puppy.service import ServiceManager, SearchService from
puppy.search.engine import Bing

from puppy.model import Query, Response

from puppy.logging import QueryLogger

# new imports

from puppy.query.modifier import TermExpansionModifier from
puppy.result.filter import ExclusionFilter

config = {
'log_dir': '/path/to/log/directory',
      # Set this to where you want the logs stored
}
sm = ServiceManager(config)
ss = SearchService(sm, 'bing_web')
sm.add_search_service(ss)
ss.search_engine = Bing(ss)
ss.query_logger = QueryLogger(ss, log_mode=0)
# Add TermExpansionModifier to SearchService
ss.add_query_modifier(TermExpansionModifier(terms='for+kids'))

# Add ExclusionFilter to SearchService

ss.add_result_filter(ExclusionFilter(terms='bad+nasty'))
query = Query('puppy')
results = sm.search_services['bing_web'].search(query).entries
for result in results.entries: print result['title']
      print result['summary']
      print result['link']
      print result['suitability'] + '\n'
```

6.2.2.2 Multiple Search Services

Whilst searching one source is useful, there are many possible situations in which a PuppyIR based service might need to search multiple sources. The simplest example, is a service that provides search suggestions alongside the main search results. The search suggestions may come from a completely different source, but, in this case, they come from a separate instance of Bing with a different source type: 'relatedSearch' (which retrieves query suggestions).

```
from puppy.service import ServiceManager, SearchService

from puppy.search.engine import Bing

from puppy.model import Query, Response

config = {}
sm = ServiceManager(config)
# As before, create a SearchService for Bing (e.g. for main results)
ss1 = SearchService(sm, 'bing_web')
sm.add_search_service(ss1)
# The default source is 'web' below is an example of using a
different source
ss1.search_engine = Bing(ss1)
# create our suggestion service
suggestions_service = SearchService(serviceManager,
'suggestion_search')
suggestions_service.search_engine = Bing(suggestions_service, source
= 'RelatedSearch')
# add SearchService to ServiceManager
serviceManager.add_search_service(suggestions_service)
query = Query('puppy')
webResults = sm.search_services['bing_web'].search(query).entries
suggestions =
sm.search_services['suggestion_search'].search(query).entries

for result in webResults:

      print result['title']
      print result['summary']
      print result['link']

for result in suggestions:
# The title is the query suggestion, i.e. for Batman a suggestion
could be: Batman Begins
      print result['title']
```

6.2.3 Exception Handling in PuppyIR

PuppyIR provides a basic set of exceptions to handle errors specific to its components. These exceptions are split between errors that occur during the Query and Result pipelines, in addition to errors that occur within a search engine wrapper.

6.2.3.1 Exception handling in the Query Pipeline

The following exceptions are available at this stage:

- **Query Rejection Error:** use this exception for when a query is rejected due to it failing one or more query filter tests. For example, if a profanity filter is used and the users query contains a swear word the query will be rejected - when catching this

exception callers should provide code to deal with this situation as no results will be returned if this occurs.

- **Query Filter Error:** use for situations in which the filter operationally failed and the filter's function cannot be realised. Callers should respond to this as if a rejection decision cannot be made.
- **Query Modifier Error:** Use for situations in which the modifier operationally failed and the modifier's function cannot be realised. Callers should respond to this as if a modification or rejection decision cannot be made. They can all be imported with the following line of code:

```
from puppy.query.exceptions import QueryRejectionError, QueryFilterError, QueryModifierError
```

An example of how to handle a query rejection error is detailed below:

```
try:
	 web_results = service.search_services['web_search'].search(query).entries
except QueryRejectionError:
	# This variable can then be used to decide to show an error or the results
	result_dict['webQueryRejected'] = True
```

6.2.3.2 Exception handling for searching within an application

The following exceptions are available at this stage:

- **Search Engine Error:** use this exception for handling issues arising from the operation of a search engine wrapper like proxy errors, the web service being down, invalid parameters etc. This is a general exception that deals with the aforementioned problems and any others that might occur.
- **API Key Error:** use this exception only if you are using search engine wrappers that require an API key (like BingV2) to ensure that the API key is supplied and has the correct field name.

They can both be imported with the following line of code:

```
from puppy.search.exceptions import SearchEngineError, ApiKeyError
```

A *'Search Engine Error'* contains the option of printing out a formatted error message; as opposed to the default, of it being outputted as one line; an example of how to handle both of the search engine exceptions and make use of the formatted print for *'SearchEngineError'* is given below:

```
# The searching code in the 'try' in simplified (full examples are found elsewhere)

formattedDesc = True

try:
	results = serviceManager.search_services['bing_web'].search(query).entries
except SearchEngineError, e:
	if formattedDesc:
		print(e.formattedStr())
	else:
		print(e) # Unformatted is the default
except ApiKeyError, e:
```

```
        print(e)
```

6.2.3.3 Exception handling in a search engine wrapper

The following two examples detail how to implement the exceptions detailed above, in a search engine wrapper, i.e. if you are extending this part of the framework.

Below is an example of how to handle an 'API key Error':

```
# Try and get the API key from config, if it's not there raise the
error
try:
      appId = self.service.config['bing_api_key']
except KeyError:
# First parameter is the wrapper name, the second is the field name
for the API key
      raise ApiKeyError('BingV2', 'bing_api_key')
```

Below is an example of how to use the 'Search Engine Error' to deal with:

- An urllib2 error, adding in extra parameters for the error message.
- A type error for some local variables.
- A general catch-all error for anything unforeseen (this enables the 'Search Engine Error' to be used in an application as a general catch all exception; yet still provide specific details).

```
try: # Omitted the code preceding the return statement see
'BingV2.py' for it in full
  return parse_bing_json('BingV2', query.search_terms, results,
sources, pos)
# urllib2 - this catches http errors due to the service being down,
lack of a proxy etc
except urllib2.URLError, e:
  raise SearchEngineError('BingV2', e, errorType = 'urllib2', url =
url)
# Check for a type error for offset or resultsPerPage
except TypeError, e:
   note = 'Please ensure that 'offset' and 'resultsPerPage' are
integers if used'

  if isinstance(offset, int) == False:
    raise SearchEngineError('BingV2', e, note = note, offsetType =
type(offset))

  if isinstance(self.resultsPerPage, int) == False:
    resultsType = type(self.resultsPerPage)
  raise SearchEngineError('BingV2', e, note = note,
resultsPerPageType = resultsType)
  raise SearchEngineError('BingV2', e, note = note)

# Catch all exception, just in case
except Exception, e:
  raise SearchEngineError('BingV2', e, url = url)
```

You can pass a 'Search Engine Error' exception as many extra parameters as required - since it uses a key/value args parameter which enables extra information, specific to your wrapper, to be added and outputted as part of the exceptions error message.

6.2.3.4 Exception handling with the Result Pipeline

The following exceptions are available at this stage:

- **Result Filter Error:** use for situations in which the filter operationally failed and the filter's function cannot be realised. Callers should respond to this as if a rejection decision cannot be made.
- **Result Modifier Error:** Use for exceptions in which the modifier operationally failed and the modifier's function cannot be realised. Callers should respond to this as if a modification or rejection decision cannot be made.

They can all be imported with the following line of code:

```
from puppy.result.exceptions import ResultFilterError,
ResultModifierError
```

6.2.4 The PuppyIR Framework Test Suite

The PuppyIR framework comes with an in-built test suite; for creating unit tests for all its components. The two main tasks are detailed below, briefly, and then discussed in the following sections.

Create a test (where <module> is the name of the Python file the test is for):

```
$ cd /path/to/framework
$ python unit.py create <module>
```

Run all tests:

```
$ cd /path/to/framework
$ python unit.py run
```

6.2.4.1 Create

Creates a skeleton test file placed at a mirror location (a structure that mirrors the framework's module structure) in the test hierarchy.

For example:

```
$ cd /path/to/framework
$ python unit.py puppy/query/filter/cool_filter.py
```

We now see, from framework's root directory, a new file at: test/puppy/query/filter/cool_filter.py - with the following auto-generated code:

```
from puppy.query.filter.cool_filter import *
import unittest
class TestCoolFilter(object):
  pass
if __name__ == '__main__':
  unittest.main()
```

It is now ready to be used and it is up to the programmer to write tests for the component in question.

6.2.4.2 Run

This command searches for all the current test cases and runs them. Issues are reported at the end; nothing is outputted if a test succeeds.

If you are using a proxy server, there are two options: either use the in-built proxy system using a ServiceManager (via it's config variable) or write a work-around for your tests or they will all fail (due to proxy errors; unless, of course, you are testing a component that does not require access to the internet via aforementioned proxy).

6.2.4.3 Example: Testing the Blacklist Filter

To provide an example, the code below shows a test for the Blacklist query filter (this rejects queries with blacklisted words in them). What this code does is check that queries with blacklisted words are actually being rejected and that valid queries are not rejected.

```
from puppy.query.filter.blacklistfilter import *
import unittest

class TestBlacklistfilter(unittest.TestCase):

  def test_main(self):

        t = BlackListFilter(terms='bad')
        self.assertTrue(t.filter(Query('hello')))
        self.assertTrue(t.filter(Query('friends')))
        self.assertFalse(t.filter(Query('bad friends')))
        self.assertFalse(t.filter(Query('bad hello')))

if __name__ == '__main__':
  unittest.main()
```

6.3 Annex C: Tutorials

The following four tutorials introduce all the main components found in the framework and provide practical examples of how to use them to create an application.

6.3.1 BaSe Tutorial: Building a PuppyIR/Django Service

The BaSe (Basic Search Engine) tutorial details how to create a Django project using the PuppyIR framework. Before starting this tutorial, please ensure that you have followed the instructions on for installing the framework and have, in addition, installed Django.

For more information on Django and a more detailed explanation of the steps detailed in this tutorial, please refer to the Django tutorial[3].

6.3.1.1 Creating a Django project and application

First, browse to the directory you want to store BaSe in and run the following command to create the project - this will create all the standard Django project files.

```
$ path/to/django/installation/django-admin.py startproject base
$ cd base
$ python manage.py runserver
```

Check it worked by loading up your browser and going to: http://localhost:8000 a standard Django page should be displayed congratulating you on creating your first Django project.

Now we will create an application within the BaSe project; called WeSe or WebSearch. It is important to note that applications, such as WeSe, cannot have the same name as the project they are part of. Run the following command from in the BaSe directory to create WeSe.

```
$ python manage.py startapp wese
```

The next step is to amend the 'settings.py' file in the BaSe folder to include the new application. Open this file and amend the installed applications section to look like this:

```
INSTALLED_APPS = (
      # All currently installed apps here
      'wese',
)
```

6.3.1.2 Configuring the WeSe application, adding a view and creating the templates

Add directory called 'template' in the BaSe folder and in 'template' create another folder called 'wese'. In this folder create a file called 'index.html', then add the following html to it:

```
<!DOCTYPE html PUBLIC '-//W3C//DTD HTML 4.01//EN'>
    <html>
      <head>
        <title>WeSe (Web Search) - a BaSe application</title>
```

[3] https://docs.djangoproject.com/en/1.3/intro/tutorial01/

```
      </head>
      <body>
        <div id='page'>
          <div id='header'>
            <h1 id='title'>WeSe (Web Search) - a BaSe
application</h1>
          </div> <!-- end header -->
          <div id='searchbox'>
            <form action='/wese/query/' onsubmit='return
validate_form(this)' method='post'>
              {% csrf_token %} <!-- cross-site request forgery
protection -->
              <input type='text' name='query' value='' id='query'>
              <input type='submit' value='Search' />
            </form>
          </div> <!-- searchbox -->
          <div id='resultbox'>
            {% block main %}{% endblock %} <!-- placeholder block for
results -->
          </div> <!-- resultbox -->
        </div> <!-- end page -->
      </body>
    </html>
```

Now we need to amend 'settings.py' in the BaSe directory to refer to this new template directory. Add the following lines of code at the top of the file:

```
import os

APP_DIR = os.getcwd()
```

This will set-up a variable with the current working directory so we can refer to the template directory without writing the absolute path. Now add the template directory so it looks like:

```
TEMPLATE_DIRS = (
        os.path.join(APP_DIR, 'templates')
)
```

The last step is to add a url for WeSe, so that Django knows which view to fetch. Load the 'url.py' file in the BaSe directory and change it so it looks like:

```
urlpatterns = patterns('', # Other URLs
        (r'^wese/$', 'wese.views.index'),
    )
```

Now add the following code to 'views.py' in the WeSe folder, this will return our index page (using the template we created earlier).

```
# Django imports

from django.template.context import RequestContext from
django.shortcuts import render_to_response

def index(request):

  '''show wese index view'''

  context = RequestContext(request)

  return render_to_response('wese/index.html', context)
```

Now go to: http://localhost:8000/wese and our index page will be displayed.

6.3.1.3 Getting and displaying search results using PuppyIR

Create a file called ‘service.py’ in the WeSe directory. This will store all our web services and set them up. Put the following code in it:

```
from puppy.service import ServiceManager, SearchService

from puppy.search.engine import Bing

from puppy.model import Query, Response

config = {} # create a ServiceManager

service = ServiceManager(config)
# create a SearchService and choose the search engine
bing_search_service = SearchService(service, ‘bing_web’)
bing_search_service.search_engine = Bing(bing_search_service)
# add SearchService to ServiceManager
service.add_search_service(bing_search_service)
```

Now we have to create a template to show our results, add a new template (in the same directory as ‘index.html’) called ‘results.html’ and add the following html to it (this template will be added to index to display the results - see Django documentation for more details on how this is done).

```
{% extends ’wese/index.html’ %}
    {% block main %}
    <p>Search Terms: <em>{{ query }}</em></p>
        {% for result in results %}
            <div class=‘result’>
            <div id=‘resulttitle’>
                    <a href=‘{{ result.link }}’>
                    <strong>{{ result.title }}</strong>
                    </a>
            </div>
            <div id=‘resultdescription’>{{ result.summary }}</div>
            <div id=‘resultlink’>{{ result.link }}</div>
            </div>
        {% endfor %}
{% endblock %}
```

We know need a view for WeSe to handle the receiving of a query, getting the results and then displaying them. Load ‘views.py’ in the WeSe directory and add the following new imports and method:

```
# From PuppyIR

from puppy.model import Query, Response

# From WeSe - get our service manager so we can search for results

from wese.service import service

def query(request):

      '''show results for query'''
```

```
    user_query = request.POST['query']

    results =
    service.search_services['bing_web'].search(Query(user_query)).e
    ntries context = RequestContext(request)

    results_dict = {'query': user_query, 'results': results}

    return render_to_response('wese/results.html', results_dict,
    context)
```

Finally, we need to add a new URL to deal with queries, load 'urls'py' from the BaSe directory and amend the code to:

```
urlpatterns = patterns('',

    # Previous URL's - these are not shown for clarity reasons
    (r'^wese/query/$', 'wese.views.query'),

)
```

Now go to: http://localhost:8000/wese and try out a few queries. Congratulations, that's you created your first PuppyIR/Django web application!

6.3.2 IfSe Tutorial: Information Foraging Search Application

6.3.2.1 Getting Started

To start this tutorial we assume that you have downloaded and installed the PuppyIR framework along with the associated Python Libraries (the later sections of this tutorial require Whoosh to be installed).

If you have not installed the PuppyIR framework or Whoosh then get everything set up before starting this tutorial.

This tutorial is designed to give you an idea of how the PuppyIR framework can be used in conjunction with Django to quickly and easily create web based search services.

To take full advantage of the framework we would highly recommend learning Python and becoming familiar with Django, however, we have also designed this tutorial so that minimal coding is required. In fact, all the lines of code needed are provided below in a step-by-step guide.

Downloading the Source Code for the Tutorial

Download the latest release of the tutorial from the PuppyIR repository:

```
$ svn co
https://puppyir.svn.sourceforge.net/svnroot/puppyir/trunk/prototypes/
ifse-tutorial
```

N.B. depending on your OS and SVN version you may need to add ' ifse-tutorial' to the end of the above command.

Running IfSe

```
$ cd /path/to/ifse-tutorial
$ python manage.py runserver
```

Visit http://localhost:8000/ifse this should bring up interface below.

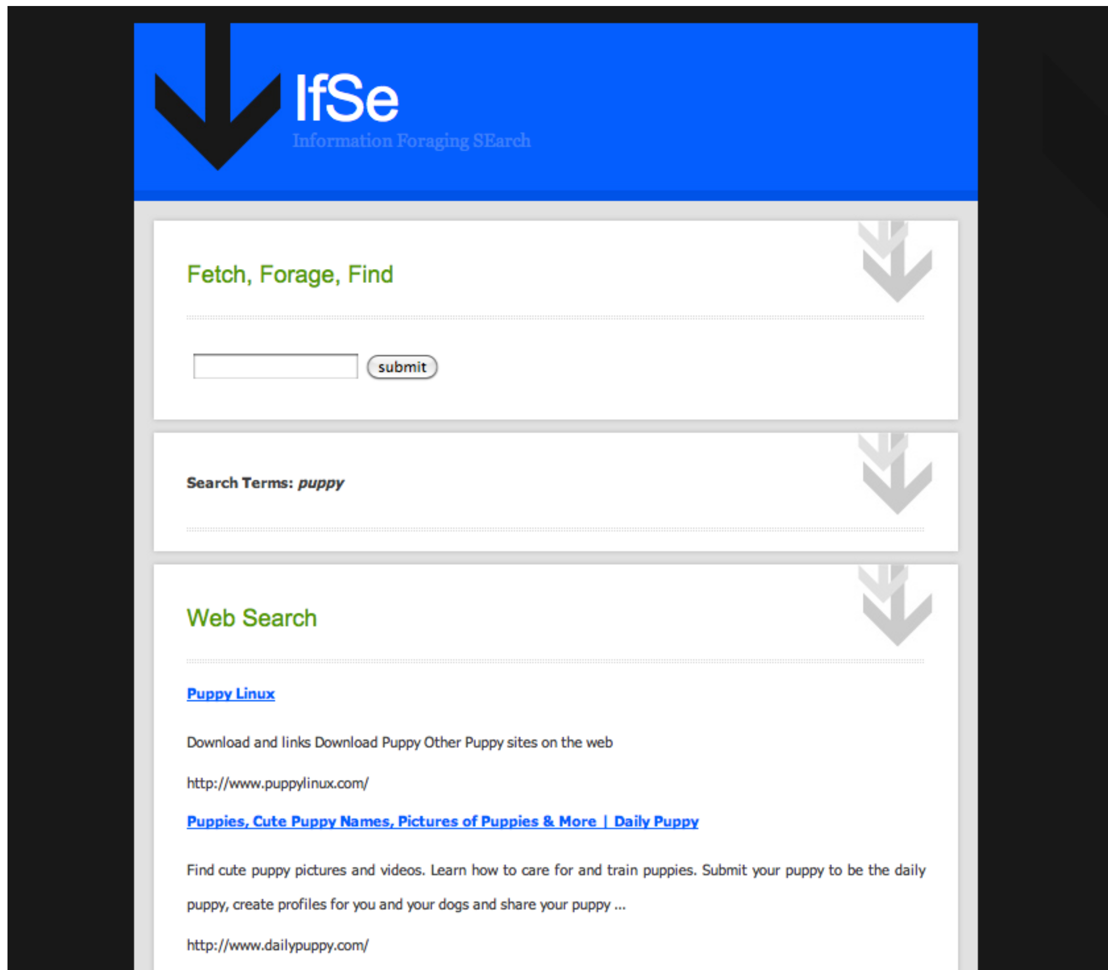

Figure 18 - *IfSe running on a local machine.*

Excellent! You now have a simple search interface that is hooked up to the Bing search API. Go on, try it out. Search for something!

While, this service means you can search the web, it doesn't record anything.

6.3.2.2 Logging Queries

Let's assume that we'd like to keep track of all the queries that users submit so that we can do a query log analysis later on.

There are a number of ways to do this. But let's do it the simple way first. Load up a code editor and open up ifse/service.py

This is where we can specify how we would like to configure our search service. We can easily add and modify search engines, query filters and result filters.

The PuppyIR framework provides a default query logger, lets include it by adding the following lines of code:

```
from puppy.logging import QueryLogger
```

This is a file based query logging. To tell PuppyIR where to store the log we need to add a 'log_dir' to the config:

```
config = {
    'log_dir': 'ifse/query_logs',
}
```

After the declaration and creation of the web_search_service, add the following line of code:

```
web_search_service.query_logger = QueryLogger(web_search_service,
log_mode=0)
```

This tells PuppyIR that you would like log queries that are submitted to this search service.

Too Easy!

Now, make sure the server is still running, i.e. python manage.py runserver and visit http://127.0.0.1:8000/ifse

Type in a few queries.

Now, go and check the directory ifse/query_logs, you should have a file in there called, web_search_query_log. This will contain a list of the queries that you have just entered. I hope you didn't type in any naughty queries!

6.3.2.3 Modifying Queries

Part of the PuppyIR project is to create child friendly services. So lets add a QueryFilter that stops naughty query terms being submitted to the search engine. To do this, we can use the BlackListFiter component that is part of the PuppyIR framework. Now add the following line of code to import it:

```
from puppy.query.filter import BlackListFilter
```

Then after the declaration and creation of the web_search_service, add the following lines of code:

```
query_black_list = BlackListFilter(0, terms = 'bad worse nasty
filthy')
web_search_service.add_query_filter(query_black_list)
```

What the Blacklist filter does, is, look at the query sent to the PuppyIR framework and check each word contained in it (the query) against the blacklist. The blacklist defines words that are not allowed (in the code example above the blacklist is populated via the second parameter; separated by spaces). If your query contains any of these words then the query will be rejected and a message displayed stating this.

Try the service now. Enter a really naughty query, like 'bad test' and see what happens. A message should be displayed stating that the query was rejected because it contained blacklisted words.

6.3.2.4 Adding Another Search Service

Instead of just returning web results, we might want to all add in other kinds of results. PuppyIR also contains various other search engine wrappers to APIs other than Bing, such as: Twitter, Yahoo, etc.

Let's create a new search service, so that we can include Twitter results as well as web results. To do this add the following line of code:

```
from puppy.search.engine import Twitter
```

And then declare and create this new SearchService and search engine, with:

```
twitter_search_service = SearchService(service,'twitter_search')
twitter_search_service.search_engine =
Twitter(twitter_search_service)
```

Don't forget to add it to the PuppyIR service manager, which is called service:

```
service.add_search_service(twitter_search_service)
```

Okay, let's try the service out now. When you enter a query now, it should return two panes of results: first, the web results and then the twitter results.

6.3.2.5 More Querying Logging

The query logger above simply dumps all the queries entered to a flat file. While this is really handy to process afterwards, it would be nice if we could index all the queries and then present similar queries as query suggestions.

To do this we need two include two components, a QueryFilter that records and indexes queries submitted to the service, and a SearchService that recommends queries. Luckily we have already implemented a simple query indexing QueryFilter that uses the Python based Whoosh indexer. The filter is called, WhooshQueryLogger, while the search engine is called WhooshQueryEngine. Let's import then into our service.py:

```
from puppy.query.filter.whooshQueryLogger import WhooshQueryLogger

from puppy.search.engine.whooshQueryEngine import WhooshQueryEngine
```

Now create the WhooshQueryLogger. It will need the full path name to the index directory. And then it needs to be added to the search_service that we wish to log, so here we can log the web_search_service:

```
whoosh_dir = os.path.join(os.getcwd(), 'ifse/query_logs/index')
whoosh_query_logger =
WhooshQueryLogger(whoosh_query_index_dir=whoosh_dir, unique=True)
web_search_service.add_query_filter(whoosh_query_logger)
```

Now, we want to provide the suggestions, so we need to create a SearchService for query suggestions and then create the WhooshQueryEngine, which also needs to know the location of the index directory:

```
suggest_service = SearchService(service, 'query_suggest_search')
whoosh_engine = WhooshQueryEngine(suggest_service,
whoosh_query_index_dir=whoosh_dir)
suggest_service.search_engine = whoosh_engine
service.add_search_service(suggest_service)
```

So let's start entering a few queries. Note, you might have to enter a few queries before you start to see recommendations appearing.

6.3.2.6 Pipelining

You might notice that if you type in 'bad query', you still get results for the twitter service. This is because we didn't add the BlackListFilter to our twitter_search_service. Do that now and make sure nothing nasty gets through.

Also, if we added the WhooshQueryLogger before the BlackListFilter then we would record all the nasty queries before rejecting the query and then start to recommend them....ooops! So it is always a good idea to pay attention to your query and document pipelines.

6.3.2.7 Give IfSe a Style

If you are interested in changing the look and feel of the application, then you can check out the html template files in templates/ifse/ within the tutorial directory, and the corresponding style sheet held in, site_media/css/

For example, open up index.html in template/ifse and change:

```
<link href='{{ MEDIA_URL }}css/concurrence/style.css'
rel='stylesheet' type='text/css'>
```

to:

```
<link href='{{ MEDIA_URL }}css/twirling/style.css' rel='stylesheet'
type='text/css'>
```

Doesn’t IfSe look prettier in pink?

Try changing perplex to combination, passageway, twirling or download any other CSS style from: http://freecsstemplates.org.

6.3.2.8 Summing Up

In this tutorial, we have only considered how to configure a service using some of the existing components within the PuppyIR framework. But it is really easy to develop your own components and customise your search service.

6.3.3 MaSe Tutorial: Mash-up Search Engine Application

6.3.3.1 Getting Started

Before starting this tutorial we assume that you have downloaded and installed the PuppyIR framework along with required associated Python Libraries (this tutorial also requires Whoosh to be installed).

If you have not installed the PuppyIR framework and/or Whoosh do so now before starting working on this tutorial.

This tutorial is designed to show the PuppyIR framework can be used to create and customise a web application, quickly, using the Django web framework. No Python experience is required to do this tutorial, as there is minimal coding involved and there are instructions regarding what coding there is (failing that, an answer file is included called ‘complete-service.py‘ which includes working code for all the tasks).

Please note that Javascript must be enabled for this tutorial to work, ask your teacher if this is the case and, if not, get them to enable Javascript.

Downloading the Source Code for the Tutorial

Now that you’ve got PuppyIR, Django and Whoosh installed it’s time to download the latest release of the tutorial from the PuppyIR repository using the following command (if you have problems with this step please ask your teacher for help):

```
$ svn co
https://puppyir.svn.sourceforge.net/svnroot/puppyir/trunk/prototypes/mase-tutorial
```

N.B. depending on your OS and SVN version you may need to add ' mase-tutorial' to the end of the above command.

Running MaSe

To run MaSe, execute the following two commands (substituting in the path to where you downloaded MaSe to):

```
$ cd /path/to/mase-tutorial
$ python manage.py runserver
```

Now, visit: http://localhost:8000/mase which should bring up the screen shown below (if you are using Internet Explorer you will not get rounded edges for your result boxes):

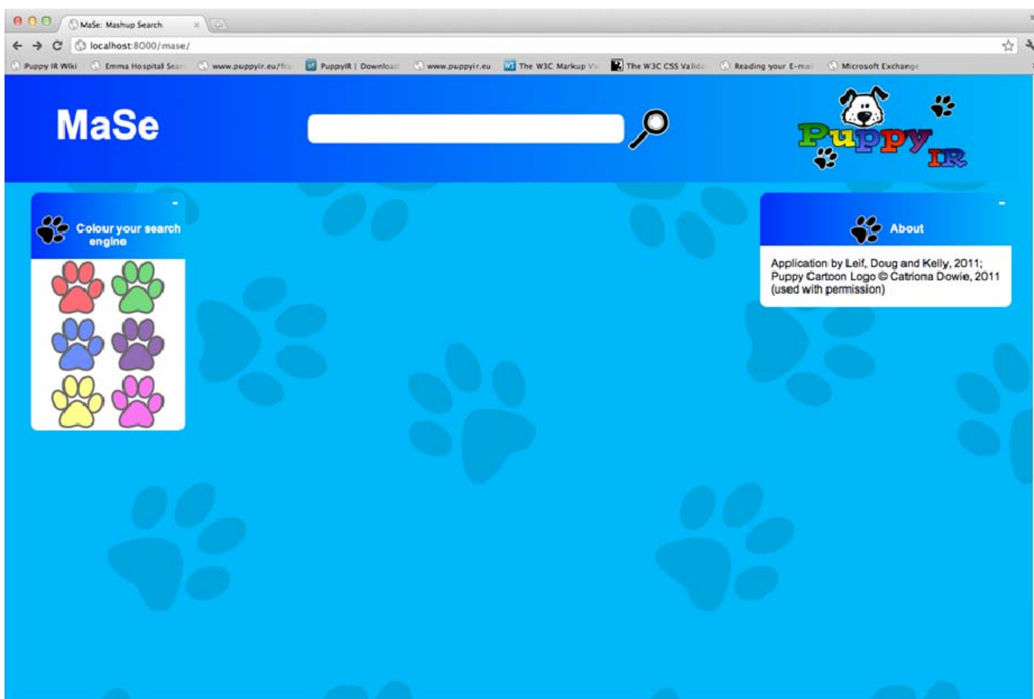

Figure 19 - *MaSe running on a local machine.*

To search for results either press enter/return in the search box or click on the magnifying glass. You can customise your search engine by:

- Clicking on the title, 'MaSe', allows you to change the name of the search engine by typing in a new name - pressing enter/return will save your search engine's new name.
- Clicking on the paw images in the 'Colour your search engine' box will change the colour theme of the search engine.
- You can also move the result boxes around on the screen (more on this in the next section).
- Minimise results by clicking on the '-' on the top right of a result box; you can maximise it again by clicking on the '+' that appears when results are minimised.

Go ahead and name your search engine and pick a new colour scheme - your new settings will be saved (using cookies; ask your teacher to enable cookies if they are disabled) so there is no need to do this every time.

6.3.3.2 Adding our first services

However, we don't have any services added yet, so, will get no results when searching. Let's fix that now by adding our first service: web results. Open the 'service.py' file in the *mase* directory and add the following lines of code, at the bottom of the file (the code comments, the lines starting with '#', detail the purpose of each line) :

```
# create a SearchService, called 'web_search'
web_search_service = SearchService(service, 'web_search')
# Set our SearchService to use the Bing search engine (it defaults to
search for web results)
web_search_service.search_engine = Bing(web_search_service)
# add SearchService to our ServiceManager (this handles all the
search services MaSe contains)
service.add_search_service(web_search_service)
```

Now refresh your browser and search for something. You should be presented with results, for your query, in a format similar to what is shown below:

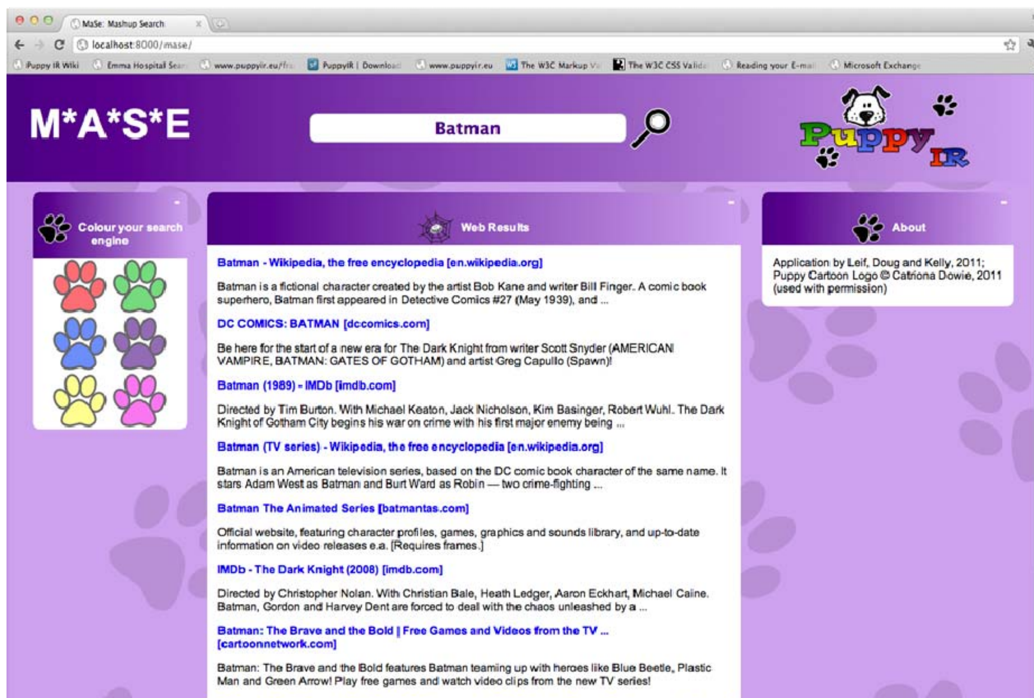

Figure 20 - *Our now customised MaSe (custom title and new colour scheme) showing web results.*

Now, let's limit the number of web results to only three, this is done by changing the line of code with 'Bing' in it to:

```
# Set the resultsPerPage parameter to 3; this limits the results the
service will return
web_search_service.search_engine = Bing(web_search_service,
resultsPerPage = 3)
```

But, it's boring just having one set of results - so lets add images as well. This is done by adding the code below (note the differences and similarities to adding web results):

```
# create a SearchService, called 'image_search'
image_service = SearchService(service, 'image_search')

# Set our SearchService to use Bing but this time with images

image_service.search_engine = Bing(image_service, source='image',
resultsPerPage = 3)
# add SearchService to our ServiceManager
service.add_search_service(image_service)
```

Go ahead and search for something, you should now see images and web results. You can also drag your results around and place them either on the left, centre, or right result columns; an example of this is shown below:

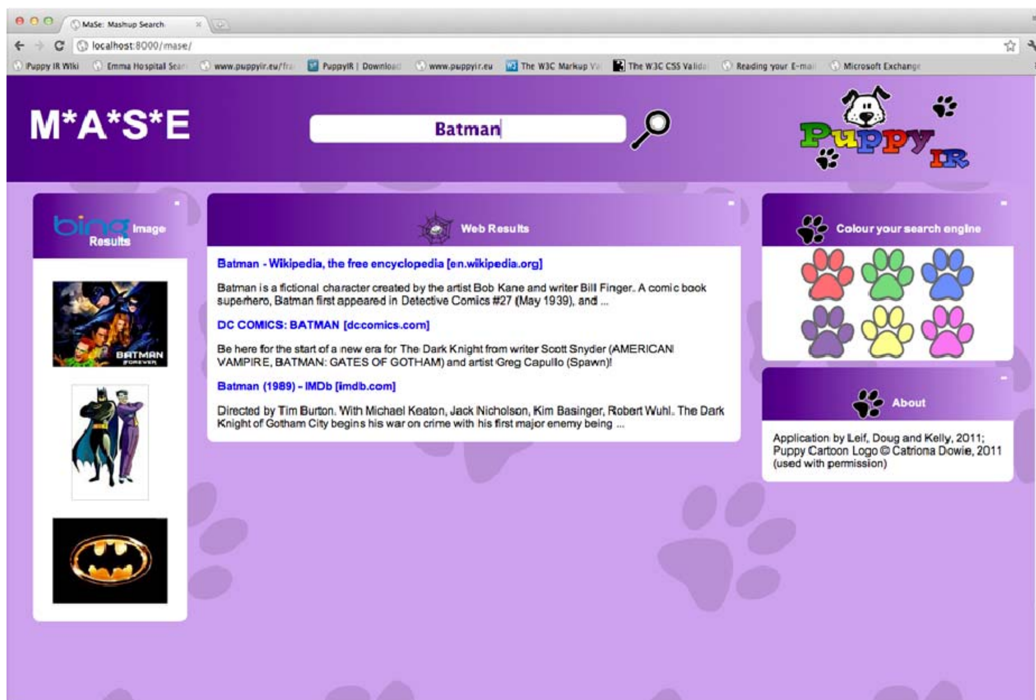

Figure 21 - *Re-arranging 'Web' and 'Image' results in MaSe.*

6.3.3.3 Extending MaSe with query logging and suggestions

Now let's add a query logger to record our queries by adding the code below just after where we created (and added) the web search service:

```
# Create a Whoosh Query Logger that records all the unique queries
whoosh_query_logger =
WhooshQueryLogger(whoosh_query_index_dir=whoosh_dir, unique=True)
# Add the Whoosh Query Logger to the web_search service
web_search_service.add_query_filter(whoosh_query_logger)
```

Next we want query suggestions, add the following lines of code to enable this feature:

```
# create a SearchService, called 'query_suggest_search'
suggest_service = SearchService(service, 'query_suggest_search')
```

```
# Use the Whoosh Query Engine to record queries
whooshEngine = WhooshQueryEngine(suggest_service,
whoosh_query_index_dir=whoosh_dir)
suggest_service.search_engine = whooshEngine
# add SearchService to our ServiceManager
service.add_search_service(suggest_service)
```

What the 'suggest_service does is to look at past queries and see if any of them contain terms from the current query. If so, it recommends those past queries as suggestions. The picture below shows query suggestions in action. Go ahead and enter a few queries now; to test if the query suggestions are working.

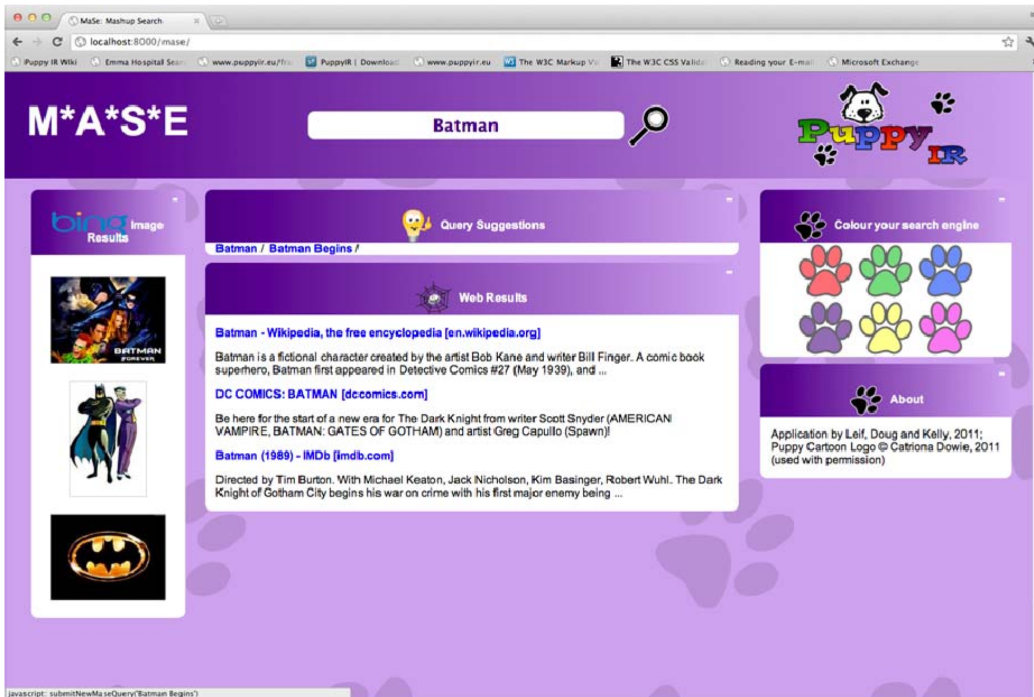

Figure 22 - *MaSe showing our now limited results for each service and query suggestions.*

6.3.3.4 Filtering and the Pipelining

Now that we've got results from three sources, lets add some filtering to stop people using your search engine to search for certain keywords. After the creation of the web_search_service, add the following lines of code to add a 'BlackListFilter':

```
# Create a blacklist filter to block queries containing the terms
below
query_black_list = BlackListFilter(terms = 'bad worse nasty filthy')
# Add our blacklist filter to the web search service
web_search_service.add_query_filter(query_black_list)
```

You will notice that if you type in 'bad query', you still get results for the image service. This is because we didn't add the 'BlackListFilter' to our 'image_service'. Do that now and make sure nothing nasty gets through.

Also, if we added the 'WhooshQueryLogger' before the 'BlackListFilter' then we would record all the nasty queries before rejecting the query and then start to recommend them as suggestions.... ooops! So it is always a good idea to pay attention to your query and document pipelines - re-order these now to stop any bad suggestions being recommended.

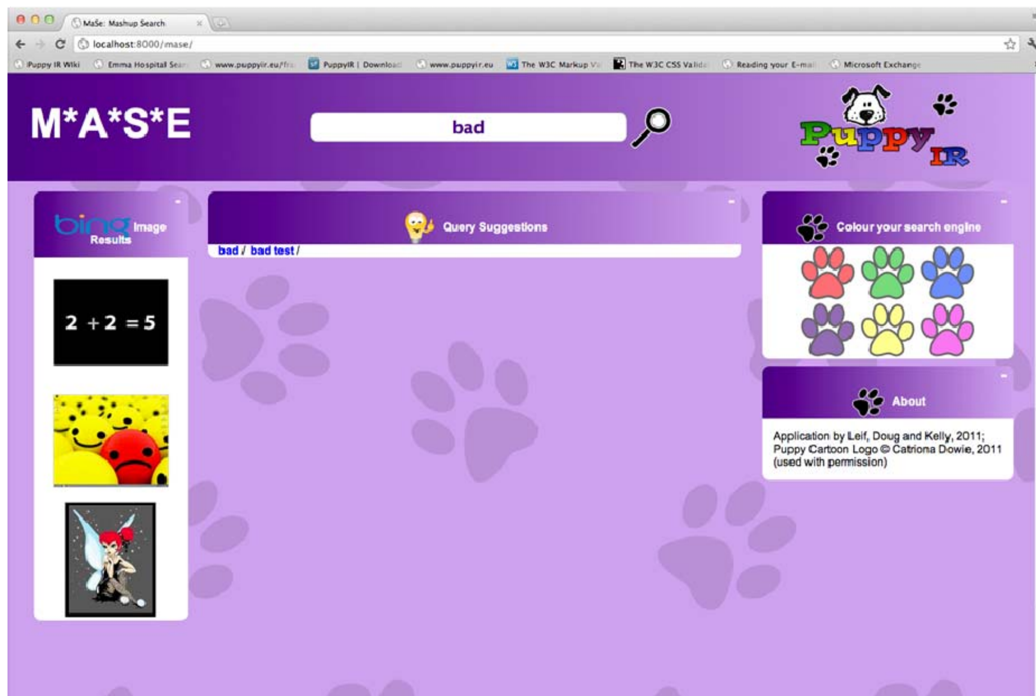

Figure 23 - *MaSe making bad suggestions and still showing image results; as in this case the filter was not added to image search*

6.3.3.5 Experimenting

Well done, that's you completed the tutorial :) - what's next is up to you, if you want to do more the following two sections contain details for suggestions for extending your search engine further.

Other Services

So far you've added images, web and query suggestions to MaSe, but there's more available.

The table below details the other options (see the code for 'web_search_service' and adapt it using the details below):

Result Source	Service Name	Class Name	Extra parameters
Wikipedia	*wiki_search*	Wikipedia	
Bing News	*news_search*	Bing	source='news'
Video (Youtube)	*video_search*	YouTubeV2	
Twitter	*twitter_search*	Twitter	

If you get stuck adding the above services then look at the file 'service-complete.py' which includes working code to add them.

You can also add in past queries with the following code (change 'web_search_service' to whatever service to want to log queries for):

```
# Log queries sent to the web search service
web_search_service.query_logger = QueryLogger(web_search_service,
log_mode=0)
```

The picture above shows what MaSe looks like with all the above services added to it with the results limited to only show the top result.

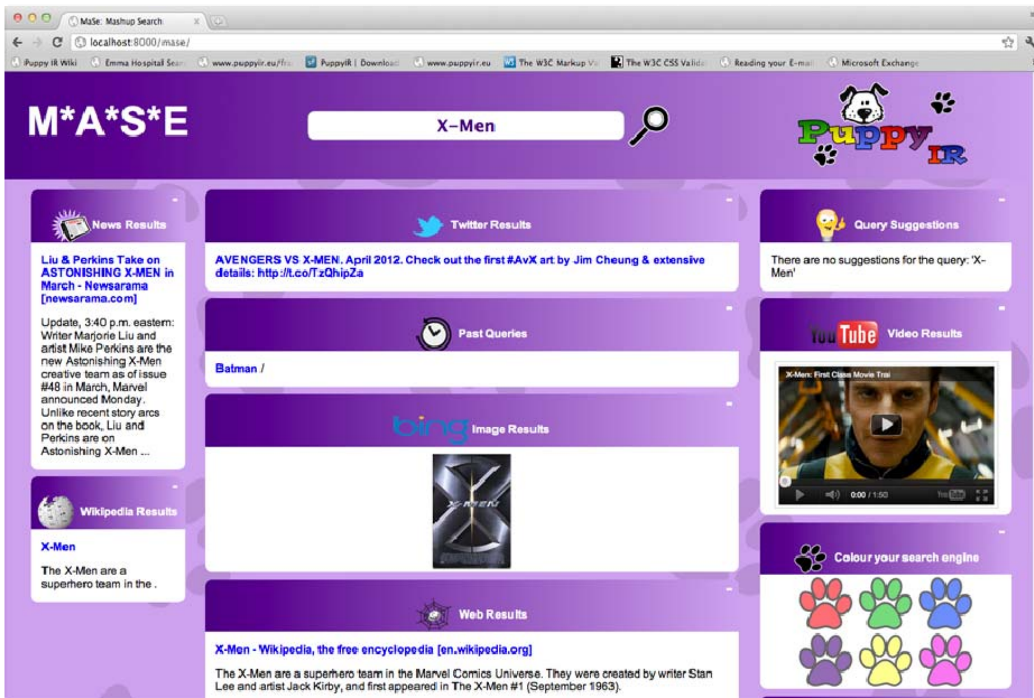

Figure 24 - *MaSe with all the different result types added to it.*

Other Parameters

There are also a few other parameters you can try out for the video and twitter services beyond 'resultsPerPage':

- Video orderBy (string), can be: 'rating', 'viewCount' or 'relevance'
- Twitter language (string), 'en' for English, 'de' for German; type (string), can be: 'mixed', 'recent' or 'popular'

6.3.4 Pipeline Tutorial: DeeSe (Detective Search)

6.3.4.1 Getting Started

If you have not installed the PuppyIR framework and/or Django, please get everything set up.

The first step is to checkout the tutorial from the PuppyIR SourceForge page and run it with the following commands:

```
$ svn co
https://puppyir.svn.sourceforge.net/svnroot/puppyir/trunk/prototypes/
deese-tutorial
$ cd /path/to/deese-tutorial
$ python manage.py runserver
```

N.B. depending on your OS and SVN version, you may need to add ' deese-tutorial' to the end of the above svn checkout command.

Now visit: http://localhost:8000/deese to see the initial version of the application. If you get stuck, at any point, during this tutorial, please consult the 'service-complete.py' file in the *'deese'* folder, this contains the 'answer' to this tutorial; along with code comments explaining each step.

6.3.4.2 DeeSe background

This tutorial is, in a sense, a companion piece to the BaSe and IfSe tutorials in that it shows how to implement similar functionality using the 'pipeline' paradigm. The scenario in this tutorial concerns a situation where the 'pipeline' paradigm is more suited the application than the 'service' paradigm.

The scenario is: you are working on an application for a team of Detectives to enable them to investigate several suspects (who have been stealing data off online websites). These suspects are well versed in electronic communication and are keeping a watch on the search history of the Detective Agency (by looking at queries sent and for their names appearing in the results the Detectives are viewing). To this end, DeeSe aims to provide the ability to search multiple sources, but have queries and results modified to prevent the names of the suspects appearing.

Therefore, for all the search services being used, one specific pipeline (for queries and results) needs to be put in place to enforce the 'lack of the suspects name' rule. Now, with the 'service' paradigm we would need to construct this pipeline for each and every source, but could we do it a different way? This tutorial details how, using the 'pipeline' paradigm, this could be accomplished.

6.3.4.3 Creating our Pipeline Service

The first step is to create a pipeline service for DeeSe (the pipeline service has already been done for you; but note how to do it) and add a search engine to it. Open up 'service.py' in the DeeSe directory and enter the following code (after the comment saying start here):

```
# Create our Pipeline Service
pipelineService = PipelineService(config, 'myPipeline')
# ----------------------- Start Here -----------------------
# Create a Bing Search Engine for news results and limit to 5 results
bingNews = Bing(pipelineService, source='news', resultsPerPage=5)

# Add Bing News to our search engine manager (this stores all our
search engines)
pipelineService.searchEngineManager.add_search_engine('News',
bingNews)
```

What this code does is to create a pipeline service and add one search engine to it (Bing News). We can now search for Bing News results using the application's searchbox. However, there is no filtering yet implemented... we should start creating our pipeline.

6.3.4.4 Setting up our Query Pipeline

Currently our query pipeline is empty (it contains no filters or modifiers) and so, will allow us to search using the suspects name; thus alerting them to the investigation. Lets stop this by constructing a query pipeline that will stop this from happening. To this end, we're going to add a query filter called 'black list filter', which will reject queries if they contain blacklisted word(s). Let's assume that the suspects are called: Bob and Nathan. Let's get coding:

```
# Let's define a variable storing the names of the suspects
```

```
suspects = 'Bob Nathan' # Separated by spaces

# Now let's create a black list query filter using the suspects
variable
blacklistF = BlackListFilter(terms=suspects)
# Add it to our pipeline service's query pipeline
pipelineService.add_query_filter(blacklistF)
```

Now, if you're confident this will work, let's try searching for 'Nathan the train job' - since one of the thefts involved a rail company. Did it work (you should get a message saying the query was rejected)? If it did, lets move onto the next stage; if not, check your code against the code above or ask for help.

6.3.4.5 But what about the results?

The other required condition was that the results returned should not contain the suspects names. For this we need to create a result pipeline to process the results. Let's add a black list modifier, what this does is 'censor' blacklisted words (by replacing them with *'s); thus, we can use this to ensure the suspects names do not appear. While we're at it, let's also add a profanity filter to stop queries containing naughty words.

```
# Let's add a Black List Modifier to alter the results
blacklistM = BlackListResultModifier(terms=suspects) # Also, as an
extra, let's stop any naughty words

profanityF = WdylProfanityQueryFilter()
# Now let's use the add filters method to add both in one go
pipelineService.add_filters(profanityF, blacklistM)
```

Try it out, can you think of queries that, while not containing the suspects names, will return results containing their names?

For the purposes of the Detective Agencies internal monitoring, all queries, both un-processed and processed (after going through the query pipeline), should be logged. Let's add a query logger to our pipeline service and set it to log processed queries (as well as the un-processed queries).

```
# Create a Query Logger and attach it to our Pipeline Service
pipelineService.query_logger = QueryLogger(pipelineService)
# Set post logging to true i.e. log processed queries (post query
pipeline)
pipelineService.postLogging = True
```

Now, search with both valid and invalid queries (i.e. ones that should be rejected). Open the log file (located in the *'deese_logs'* directory) and take a look at your query history. Note that queries that were not rejected are logged twice (un-processed and processed) and that rejected queries are only logged once. This is because when a query is rejected the search is aborted so there never is a processed query. Also, since we never added any query modifiers the processed queries are the same as their un-processed counterpart.

6.3.4.6 Let's add some new Search Services

Of course, just searching Bing News does not really offer the multiple search services required; let's add Wikipedia and Bing Web as well:

```
# Create Bing Web and Wikipedia Search Engines (again, limiting to 5
results)
bingWeb = Bing(pipelineService, source='web', resultsPerPage=5)
wikipedia = Wikipedia(pipelineService, resultsPerPage=5)
# Add our new Search Engines to our Search Engine Manager
```

```
pipelineService.searchEngineManager.add_search_engine('Web', bingWeb)
pipelineService.searchEngineManager.add_search_engine('Wikipedia',
wikipedia)
```

Now search for something, notice that the results appear for all of these search engines using the name we supplied: Web, Wikipedia, News as the title. Also, we did not need to alter 'views.py' to get results from the new search engines (which you would have to do if using the 'service' paradigm). This is because we are using the 'searchAll' method call; you could also search them one by one using 'searchSpecific' - which makes use of the name of the search engine. Due to this, we can easily add and remove search engines as required.

As an extension task, to allow you to fully understand how DeeSe allows new search engines to be added, have a look at the 'index.html' template. The Django template language code is fully commented, explaining the purpose of each line and how the results of each service are accessed & displayed (also note how the template only shows details about a search engine if it returned one or more results). This is an example of how the overall results dictionary can be processed by an application.

6.3.4.7 Next Steps

Congratulations, that's you completed the tutorial, However, there is more you could do with DeeSe:

- If you look in 'views.py' you will notice that there is code for that looks for a variable called 'offset' as well as a query. This is to allow for browsing between pages of results, what changes/additions would you have to make to implement this? [Hint: you will need to change the template]
- Styling, perhaps you could add more images and alter the style to suit the Detectives more?
- Extending the pipeline, what else could you add to DeeSe in terms of both the query and result pipelines?
- Are there any other search services you could add: videos, images? [Hint: you will need to alter the template and 'views.py']

6.3.5 Tutorial: Configuration editor

It is possible to describe tutorials explaining the use of the configuration interface from three points of view:

- The search editor or system administrator
- A developer using the Open Search Framework
- A developer using the Open Search Framework inside Django.

6.3.5.1 Search configuration

3 basic elements could be found in the framework:

- The search engine
- The filters or modifiers for the query (the query pipeline)
- The filters or modifiers for the results (the result pipeline)

It is important to remark that any of these elements could have some parameters.

In the Django model, 4 kinds of elements could be represented:

- Search engine; query filter or search engine list. The list is initialized by a script, taking all the available filters in the correspondent directory automatically.

- Parameters; A parameter is composed by a key or parameter name (for instance terms) and a value, for instance, Cartoon.
- QueryFilterOrder or ResultFilterOrder; this is the actual 'order' or command to apply a filter. The order is composed of a number, in order to choose the sequence of filters in the pipeline, the filter to use and some parameters (if needed).

This is represented by a Django model (see file puppy/interface/configuration/model.py in the source code). This way, it is possible to use the Django adminintration interface to change the data, like with any other Django model.

In order to start the admin application, the 'search editor' should go to puppy/interface and then execute Django there

```
python manage.py runserver 8008
```

and then, the connection should be made by using any browser (explorer, firefox) to `localhost:8008/admin/configuration` (user:root, passwd: root by default)

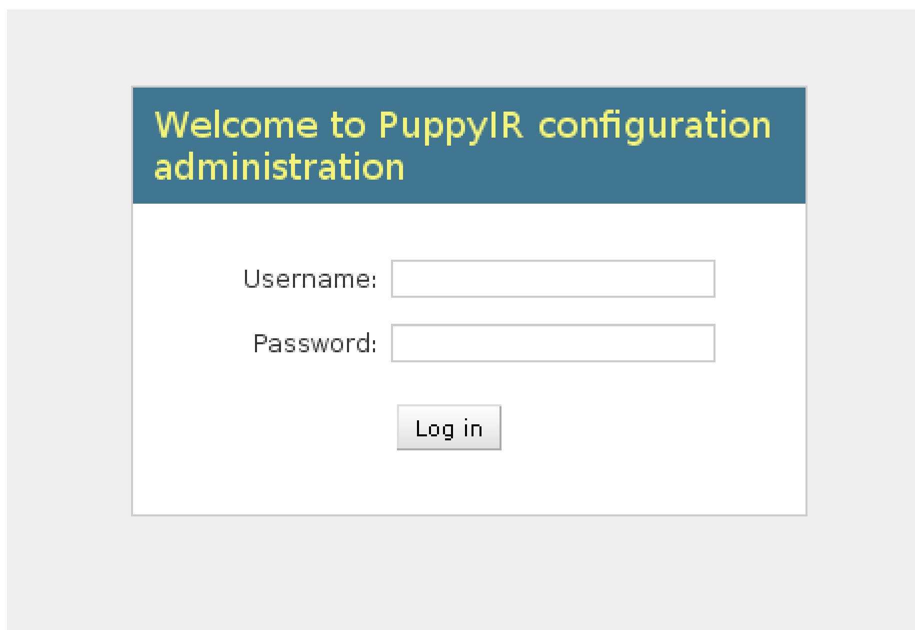

Figure 25 – *Login screen*

In previous versions, access to the lists of possible search engines or filters should be found here. These lists are not necessary; they are filled automatically to fill them using a script: main_loadTables.py, placed in the root directory.

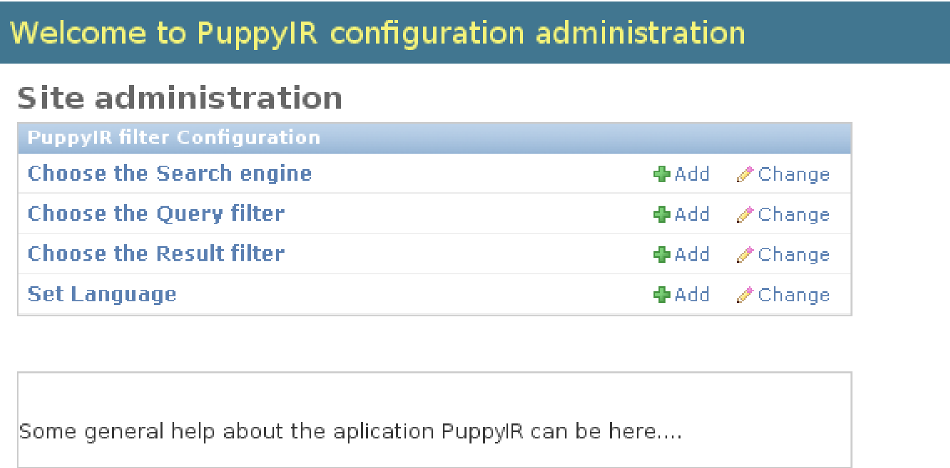

Figure 26 – *Main screen*

If the search editor needs to edit Search Engine Used, for instance, he/she could choose Bing. One and only one search engine (in the current implementation) should be chosen. The search editor could also add parameters to the engine, for instance if we want to pass 'site=www.myhospital.com', we put site as key and www.myhospital.com as value. He/she could add as many parameters as needed.

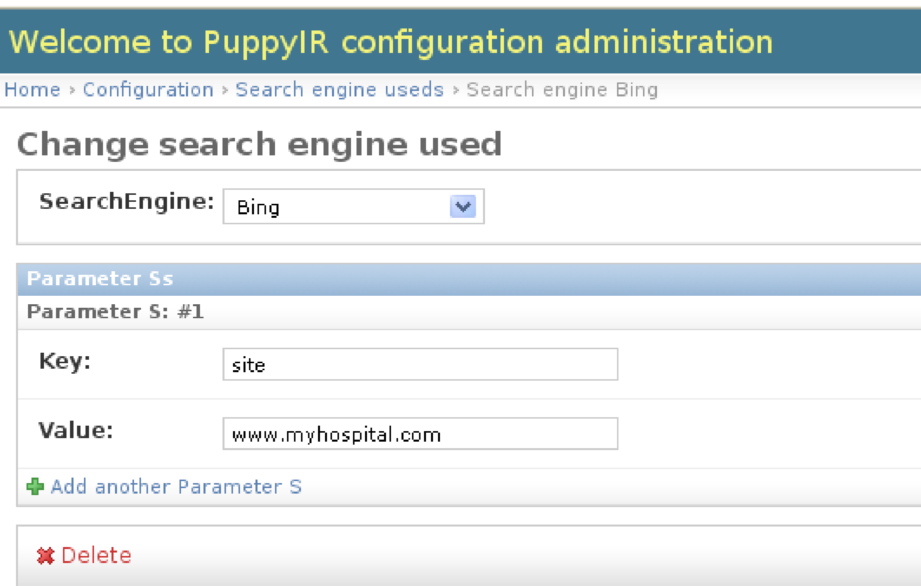

Figure 27 – *Adding a search engine*

Filters could be added to the search editor by clicking, for example, in Query Filter Orders. A list of current applied filters appears. Filters could be edited or added just by clicking on it. Here, the step in the pipeline should be chosen (for instance, 1 means that this filter will be the first to be applied), the filter to be applied (for instance, TermExpansionFilter) and parameters. As in the search engine case, terms='Cartoon' is introduce writing down 'term' as key and 'Cartoon' as value.

Figure 28 – *Adding a filter.*

For a result filter, the process will be exactly the same.

6.3.5.2 The implementation and a Django-independent applications (for developers)

The main point here is that parts of Django (as models) could be used outside the Django framework.

Nearly all implementation is centred on the file **configuredseach.py** and the directory **interfaces**. The other files in the framework are independent of the configuration interface.

There is a good example of use in the file **main_search_conf.py**. This example is similar to the general example **main_search.py**. As could be seen, from the developers point of view the use is now simpler, the code is considerably shorter than in the general example.

Now, import ConfiguratedSearch is needed. It's not necessary to import anything related to Django here.

```
from puppy.service import SearchService, ServiceManager,
ConfiguratedSearch
```

It is only needed to pass to the function the initialization values, the same as in the normal use of the framework.

```
cf = {
  #'proxyhost': 'http://wwwcache.gla.ac.uk:8080',
  'log_dir': 'logs',
  'google_api_key': 'ABQIAAAAFAoRzdMk5GDWWqERtyknf2g',
```

```
  ‘yahoo_api_key’: ‘JYjcWATV34E4S9u.wOcYZMHJCC3pE-’,
  ‘puppy_path’: ‘d:/documents/puppy_sf/framework_conf/’
}
```

And in only a line, the configured search service is created:

```
cs = ConfiguratedSearch(cf, ‘bing_web’)
```

To pass the cf and a name is needed, only for identification.

To pass more parameters or construct the pipeline is not needed, but it’s not forbidden. The object constructed is exactly the same and more things could be added without problems.

Regarding the implementation in the Open Framework, as written above, it is centred in the file service/configuredsearch.py.

The most difficult part is setting the environment variables for using the Django models outside Django, and then import the ‘settings’ file from the ‘interface’ file

```
APP_DIR = os.getcwd( )
sys.path.append(os.path.join(APP_DIR, 'puppy/interface'))
import settings
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
```

After this, we can import the models

```
from django.db import models

from puppy.interface.configuration.models import *
```

From this point, the configuration models could be used the same as from inside a typical Django application. So access to the database tables described in the previous point is enabled. The values are taken and the objects for the search engine are dynamically created, as well as the query filters and modifiers, and the result filters and modifiers. Reflection is used for the creation of the objects referred by name in the database. Then it is added to the service.

```
ptr_func = globals()[entry.queryFilter.queryFilter]

self.search_service.add_query_modifier(ptr_func(order=entry.numOrder,
**dictPar))
```

6.3.5.3 Using the configuration tool inside an existing Django application

The main objective is to allow the co-existence of the configuration module and its model and database, with an existing model and databases of the Django application.

So there was necessary to be careful about the ‘python path’ and where Django takes the elements in different parts of the program. In practice, the idea is really similar to the previous case, but the coexistence of the two different models could be problematic.

The suggestion application IfSe (described above) will be taken as example. The example modified version is in a branch called ‘demo_hospital_conf’ in the SVN repository. In the suggestion directory, the files **service.py** and **views.py** are changed.

In views.py, a ConfiguratedSearch is created, instead of using the previous version.

```
cs = ConfiguratedSearch(config, 'site_search', dictParS = {'site':sites.getSites()})

service = cs.sm
```

As told before, the object that is created is exactly the same as the previous object, so it is possible to add additional operations just as before. In this case, the created service inside the object could be taken and used to add an additional search object.

```
service.add_search_service(query_suggest_search_service)
```

The file **views.py** is also a bit modified. The createService() function is called from **service.py** every time that the user makes a search, in order to reflect the last changes in configuration. This is not very efficient and an alternative is being looked for.

In the case of IfSe, the implementation is relatively simple because IfSe does not use models. For an application using models, the easier way is mixing the model of the configuration part with the model of the application. This break the isolation, the norm of a good library, but it is the more practical way.

6.4 Annex D: Extending the Framework

6.4.1 Extending the Query Pipeline

This section details how to add new Query Filters and Query Modifiers.

Note: there is an optional parameter for both called 'order' to indicate the precedence of the filter or modifier in question.

6.4.1.1 The Query Operator base class

Both filters and modifiers extend the base class QueryOperator:

```
class _QueryOperator(object):
  '''Abstract class for query filters.'''
  def __init__(self, order=0):
      self.name = self.__class__.__name__ self.description = ''
      self.order = order
```

This contains the attributes common to both filters and modifiers: name, description and order (this defines the order in which a filter or a modifier is executed in their respective pipelines).

Note: this class is detailed for reference only, since it is not expected that this base class will be modified when extending PuppyIR.

6.4.1.2 Creating new Query Filters

All Query Filters must extend the base class QueryFilter:

```
class QueryFilter(_QueryOperator):
  '''Base class for query filters'''
  def __call__(self, *args):
    return self.filter(*args)
  @ensure_query
  def filter(self, query):
    raise NotImplementedError()
```

The filter method *must* return either: true or false - depending upon whether, or not, the defined criteria is met.

For example, a BlackListFilter that rejects queries if they contain blacklisted words:

```
import string
from puppy.query import QueryFilter
from puppy.model import Query

class BlackListFilter(QueryFilter):

  def __init__(self, order=0, terms=''):
    super(BlackListFilter, self).__init__(order) self.description =
'Rejects queries containing any blacklisted terms.'
    self.terms = set(terms.lower().split())

def filter(self, query):
    '''
```

```
      Rejects queries containing any of the defined blacklisted
terms.

      Parameters:
      * query (puppy.model.Query): original query
      Returns:
      * query (puppy.model.Query): filtered query

      '''

  original_terms = set(query.search_terms.lower().split()) return not
(original_terms & self.terms)
```

6.4.1.3 Creating new Query Modifiers

All Query Modifiers must extend the base class QueryModifier:

```
class QueryModifier(_QueryOperator):

  def __call__(self, *args):

      # shortcut for modify
      return self.modify(*args)

 @ensure_query
  def modify(self, query):

    raise NotImplementedError()
```

The modify method *must* be passed and also return a query object.

For example, a TermExpansionModifier that appends extra terms onto a query for example adding ‘for kids’ to each query:

```
from puppy.query import QueryModifier

from puppy.model import Query

class TermExpansionModifier(QueryModifier):

  '''Expands original query terms with extra terms.'''

  def __init__(self, order=0, terms=''):
    super(TermExpansionModifier, self).__init__(order)
    self.description = 'Expands original query terms with extra
terms.'
    self.terms = terms

  def modify(self, query):
    '''
    Expands query with additional terms.
    Parameters:
    * query (puppy.model.Query): original query
    Returns:
    * query (puppy.model.Query): expanded query
   '''
    query.search_terms = ' '.join([query.search_terms, self.terms])
return query
```

6.4.2 Extending the Result Pipeline

This section details adding new Result Filters and Result Modifiers.

Note: there is an optional parameter for both called 'order' to indicate the precedence of the filter or modifier in question.

6.4.2.1 The Orderable base class

Both filters and modifiers extend the base class Orderable:

```
class Orderable(object): def __init__(self, order=0):

    self.order = order
    self._init()

    def _init(self):

      raise NotImplementedError()
```

This contains the attributes common to both filters and modifiers: the order (this defines the order in which a filter or a modifier is executed in their respective pipelines).

Note: this class is detailed for reference only, since it is not expected that this base class will be modified when extending PuppyIR.

6.4.2.2 Creating new Result Filters

All Result Filters must extend the base class ResultFilter:

```
class ResultFilter(Orderable): '''Abstract result filter.'''

  def _init(self):

    self.name = self.__class__.__name__
    self.description = ''

  def __call__(self, *args):

    return self.filter(*args)

  def filter(self, results):

    ''' Return a boolean of whether this filter succeeded. '''

    raise NotImplementedError()
```

The filter method *must* return either: true or false - depending upon whether, or not, the defined criteria is met. For example, a ProfanityFilter that rejects results if their title does not pass the WDYL services test (this is a Google web service):

```
from puppy.result import ResultFilter

from puppy.query.filter.profanity_filter import WdylProfanityFilter
as WQF

import urllib
```

```
class WdylProfanityFilter(ResultFilter):

  ''' Filters results with profanity in them by using the wdyl
service.'''

  def __init__(self, order=0):
    super(WdylProfanityFilter, self).__init__(order)
   self._filter = WQF()

  def filter(self, results): # Go through each result and check each
field doesn't contain words in the exclusion list
    for result in results:
      if self._filter(result['title']):
        yield result
```

6.4.2.3 Creating new Result Modifiers

All Result Modifiers must extend the base class ResultModifier:

```
class ResultModifier(Orderable):

  ''' Change result. '''

  def _init(self):
    self.name = self.__class__.__name__
    self.description = ''

  def __call__(self, *args):
    return self.modify(*args)

  def modify(self, results):
    ''' Return a result, modified. '''
    raise NotImplementedError()
```

The modify method *must* be passed and also return a response object.

For example, a modifier called TitleBlackListModifier that replaces blacklisted words in the title with ***.

```
import string

from puppy.result import ResultModifier

class TitleBlackListModifier(ResultModifier):

  '''

  Modify processes result entry content and replaces blacklisted
words
  Options:
  * order (int): modifier precedence
  * terms (str): terms that, if appearing in the result, will be
replaced with ***
  '''
def __init__(self, order=0, terms=''): '''

      Constructor for BlackListResultModifier.
      Parameters:
      * order (int): filter precedence
```

```
        * terms (str): separated by + characters
        '''
        super(TitleBlackListModifier, self).__init__(order)
        self.info = 'Modify search results based on a blacklist.'
        self.terms = terms
        self.black_list = ' '.join(filter(str.isalpha,
terms.replace('+', ' ').lower().split()))


def apply_black_list(self, input_string): '''

        Replaces words in black list for *** characters.
        Parameters:
        * black_list_string: string with words included in the black
list
        * input_string: string with words separated by blank spaces
Returns: * ouput_string: string of words separated by blank spaces
which words included in the black list has been replaced by ***

''' input_list = input_string.split() output_string = input_string

for input in input_list:

  try:

    input_filtered = ''.join(filter(str.isalpha,
list(input.lower())))
  except TypeError:

    tmp = input.encode('utf-8').lower()
    input_filtered = ''.join(filter(str.isalpha, list(tmp)))
  if input_filtered in self.black_list:

   if input_filtered not in ' ':

   output_string = output_string.replace(input, '***')
 return    output_string

def modify(self, results):
     '''

        Filters the results according to black list -
        censoring any blacklisted words occurring in results.
        Parameters:
        * results (puppy.model.Opensearch.Response): results to be
filtered
   Returns: * results_returned (puppy.model.Opensearch.Response):
filtered results     '''
  for result in results:
    result['title'] = self.apply_black_list(result['title'])
  yield result
```

6.4.3 Adding new Search Engine Wrappers

This section details adding new search engine wrappers. Firstly, every wrapper must extend the base class SearchEngine.

6.4.3.1 The SearchEngine base class

This base class defines the standard attributes common to all search engine wrappers. It also provides the facility to use search engines within a proxy server if this is required. The key aspect is that the search method must be overwritten by any derived classes.

```
import urllib2

class SearchEngine(object):

  '''Abstract search engine interface.'''
def __init__(self, service):

    '''

    Constructor for SearchEngine.
    Parameters:
   * service (puppy.service.SearchService): A reference to the parent
search service

   * options (dict) a dictionary of engine specific options

   '''

    self.name = self.__class__.__name__
    self.service = service
    self.configure_opener()


def _origin(self):

  ''' This defines the default origin for results from a search
engine '''
  return 0

def configure_opener(self):

'''Configure urllib2 opener with network proxy'''
  if 'proxyhost' in self.service.config:
    proxy_support = urllib2.ProxyHandler({'http':
self.service.config['proxyhost']})

    opener = urllib2.build_opener(proxy_support)
else:
    opener = urllib2.build_opener()

urllib2.install_opener(opener)



def search(self, query, pos=1):

  '''

  Perform a search.
  Parameters:
  * query (puppy.model.Query): query object
  * offset (int): result offset
  Returns:
```

```
  * results (puppy.model.Response): results of the search
  '''
  pass
```

6.4.3.2 Creating a new Search Engine wrapper

When adding new search engine wrappers, the base class (SearchEngine) will be used and extended to process results from the new service. The Picasa (an online image sharing website) wrapper is included below to illustrate how to go about adding new wrappers.

The search method must be passed a Query object and return a Response object (these are models defined in the PuppyIR framework).

```
import urllib2

from puppy.search import SearchEngine

from puppy.model import Query, Response

class Picasa(SearchEngine): '''

  Picassa search engine.
  Parameters:
  * resultsPerPage: select how many results per page
  '''
def __init__(self, service, resultsPerPage=8):

  self.maxResults = maxResults

  super(Picasa, self).__init__(service)

def search(self, query, offset):

    '''

    Search function for Picasa.
    Parameters:
    * query (puppy.model.OpenSearch.Query)
    Returns:
    * puppy.model.OpenSearch.Response
    Raises:
    * urllib2.URLError
'''

    userQuery = urllib2.quote(query.search_terms)
    url =
'https://picasaweb.google.com/data/feed/api/all?q={0}&kind=photo'.for
mat(userQuery)
    # Add in the resultsPerPage parameter
    url += '&max-results={0}'.format(self.resultsPerPage)
try:
  data = urllib2.urlopen(url) return Response.parse_feed(data.read())
except urllib2.URLError, e:
  print 'Error in Search Service: Picassa search failed
```

6.4.3.3 Origin of the results

Results from a search engine are, generally, either 0 or 1 indexed depending upon the service in question. To account for this, as shown in the code of SearchEngine, there is an origin defined and each service uses the following code to work out which page to use (in the URL parameters):

```
pos = self._origin() + offset
```

The default is '0' and so, if a search engine is 1-indexed, for example, the search engine wrapper must override the origin in SearchEngine with its own version (the code for pos is unchanged):

```
def _origin(self):
  ''' This SearchEngine is 1-indexed so override the default'''
  return 1
```

6.4.3.4 Json and other formats

The standard method, as detailed above, is for wrappers to parse RSS/Atom feeds to retrieve the results. However, not all API's return results in this format and so, if other formats are used the wrapper itself will need to parse them. The result of this parsing must be a response object with all the standard fields required by the OpenSearch standard.

For examples of how to do this, consult the code in the following wrappers:

- **JSON:** the Guardian and Yahoo! wrappers.
- **XML:** the Wikipedia and Simple Wikipedia wrappers.

It is possible to describe tutorials of the use of the configuration interface from three points of view:

- The search editor or system administrator
- A developer using the Open Search Framework
- A developer using the Open Search Framework inside Django.

6.4.4 On Filters, Modifiers and Query Logging

6.4.4.1 Paradigm 1: One Pipeline, One Search Engine

Within each of these pipelines (query and result) there are both filters and modifiers. Filters are executed first and then, following this, the modifiers are executed.

The distinction between a filter and a modifier is as follows:

- **Filters:** these reject or accept a query, or result, based on a defined criteria. For example a blacklist filter rejects queries containing one or more blacklisted words.
- **Modifiers:** these change the content of a query, or result, based on a defined behaviour. For example, appending 'for kids' to every query.

There are two points at which queries can be logged: before the query goes through the query pipeline and after (i.e. un-processed and processed). The default is to log queries before processing - if a query logger has been added. The code below shows how to add a query logger and set it so that processed queries are logged, in addition to un-processed ones:

```
from puppy.logging import QueryLogger
```

```
from puppy.service import ServiceManager, SearchService

config = {'log_dir': '/path/to/log/dir'}
# Sets the log directory
sm = ServiceManager(config)
ss = SearchService(sm, 'bing_web')
sm.add_search_service(ss)
ss.search_engine = Bing(ss)

# Assign QueryLogger to SearchService
ss.query_logger = QueryLogger(ss)
ss.postLogging = True # Activate post-pipeline query logging
```

Within the query and result pipelines there are both filters and modifiers. Filters are executed first and then, following this, the modifiers are executed.

6.4.4.2 Paradigm 2: One Pipeline, Many Search Engines

The distinction between a filter and a modifier is as follows:

- **Filters:** these reject or accept a query, or result, based on a defined criteria. For example a blacklist filter rejects queries containing one or more blacklisted words.
- **Modifiers:** these change the content of a query, or result, based on a defined behaviour. For example, appending 'for kids' to every query.

There are many different filters and modifiers available for both of these pipelines.

There are two points at which queries can be logged: before the query goes through the query pipeline and after; i.e. un-processed and processed. The default is to log queries before processing - if a query logger has been added. The code below shows how to add a query logger and set it so that processed queries are logged, in addition to un-processed ones:

```
from puppy.logging import QueryLogger
from puppy.pipeline import PipelineService

config = {'log_dir': '/path/to/log/dir'}

# Sets the log directory
pm = PipelineService(config)
pm.query_logger = QueryLogger(pm)
pm.postLogging = True # Activate post-pipeline query logging
```

References

[1] Annex 1: Description of Work, PuppyIR, 2008

[2] D1.2 – Agreed User Requirements and Scenarios', PuppyIR, 2009

[3] D1.3 – Agreed Technical Requirements, PuppyIR, 2009

[4] D3.1 – Report of Data Pre-processing, PuppyIR, 2009

[5] D4.1 – Specification Report, PuppyIR, 2010

[6] D4.2 – Design Report, PuppyIR, 2010

[7] D4.3 – Report on Implementation and Documentation, PuppyIR, 2010

[8] D4.4 – Release of the Open Source Framework V1.0, PuppyIR, 2011

[9] D4.5 – Report on Design and Specification Changes, PuppyIR, 2011

[10] Django Project (Web Application Framework): http://www.djangoproject.com/

[11] D4.6 – Release of the Open Source Framework V2.0, PuppyIR, 2011

[12] D7.3 – Hospital Demonstrator – Version 1.0, PuppyIR, 2011

[13] D7.4 – Hospital Demonstrator – Version 2.0, PuppyIR, 2012

[14] D7.2 – Museon Demonstrator – Version 2.0, PuppyIR, 2012

[15] 'Web Search Query Assistance Functionality for Young Audiences', Carsten Eickhoff, Tamara Polajnar, Karl Gyllstrom, Sergio Duarte Torres and Richard Glassey, Lecture Notes in Computer Science, 2011, Volume 6611/2011, 776-779, DOI: 10.1007/978-3-642-20161-5_92. Available: http://www.springerlink.com/content/f51364260247hqr1/