PuppyIR Documentation

Release 3.0

The PuppyIR Development Team

August 29, 2012

CONTENTS

This documentation describes the PuppyIR project's open source, Python based, framework. This framework provides an open source environment for building information services, specifically for children, using a variety of tools, search engine wrappers, filters and more.

In order to best describe the framework and its uses, the documentation is split up into several sections:

- the **'about'** section, which details the design of the framework and how the various components relate to each other;
- the **'using the framework'** section details various aspects pertaining to the usage of the framework;
- the **'tutorials'** section, which provides practical examples of using the framework for a variety of audiences;
- the **'extending'** section details how the framework can be extended to add new filters, modifiers and search engine wrappers;
- the **'appendices'** section which details various supplementary materials that further expand on the other sections including an FAQ (frequently asked questions);
- and, finally, the **'API reference'** which details the components in framework (including details about their parameters etc).

CHAPTER

ONE

ABOUT THE FRAMEWORK

1.1 Overview and background of the PuppyIR Framework

The framework is part of the PuppyIR project [1], as 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:

1. Simplify the process of building interactive IR services;
2. provide a disparate and extensive suite of components, specifically tailored for children;
3. incorporate current research findings in children's IR;
4. 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;
5. and, to provide extensive documentation [this document] with tutorials detailing how to use, extend and customise all the different parts of the framework.

1.1.1 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 (for a full list, of all the individual components in these sections, please see consult the *PuppyIR API Reference*).

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/

Architectural Paradigms

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

1. **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. See: *Paradigm 1 - One Pipeline, One Search Engine* for a more in-depth discussion of this paradigm.
2. **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 the 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. See: *Paradigm 2 - One Pipeline, Many Search Engines* for a more in-depth discussion of this paradigm.

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 a pipeline service returns 'response' objects in a slightly different way).

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:

1. **Un-processed**: the query passed to the service in question before it goes through the query pipeline.
2. **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).

Testing and Exception Handling

The PuppyIR framework comes with a variety of custom exceptions for its components and also a unit test suite. These are discussed, in-depth, in *Exception Handling in PuppyIR* and *The PuppyIR Framework Test Suite*.

Filters and Modifiers

The *PuppyIR API Reference* contains the full details of all the varied filters and modifiers that come with the PuppyIR framework. Examples include: a query filter, that rejects queries containing profanity; a suitability filter, that evaluates

the suitability of a web result (for children) then accepts or rejects it based on a minimum score and a term expansion modifier, that adds extra terms to a user's query (like 'for kids' to skew results in certain ways).

1.1.2 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). For more details of the implementation and usage of these wrappers please refer to the *PuppyIR API Reference*.

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.

Book Services

Service Name	Description
Google-Books	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).

Image Services

Service Name	Description
Bing and **BingV2**	Allows for the retrieval of images using Bing's search API, including options like only get widescreen images.
Flickr	Retrieves images from the Flickr web service, including lots of details like geotags and more.
Picassa	Retrieves images from the Picassa web service.

Information Services

Service Name	Description
Wikipedia	Allows 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 it is ideal for providing a short description for children.
Simple Wikipedia	An alternate version of the above wrapper, using the Simple Wikipedia variant of the Wikipedia API.

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

Service Name	Description
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 'Edinburgh' 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.
YouTube V2	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.

Movie Services

Service Name	Description
Rotten Tomatoes	Allows for the retrieval of details about movies like: the cast and aggregated review score etc.

Music Services

N.B. **YouTube** and **YouTubeV2** are also, arguably, a music based services due to the large proportion of music based content.

Service Name	Description
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.
Sound-cloud	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.

News Services

These wrappers provide the ability to search for news stories.

Service Name	Description
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.

Social Network and Social News Services

Service Name	Description
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.

Spelling Suggestions and Dictionary based results

Service Name	Description
BingV2	Using the *spell* source type 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).
Web Spell Checker	Allows for searching for spelling corrections in a variety of languages - there is no extra information returned however just the spelling correction suggestion.

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.

Service Name	Description
BingV2	Allows for the retrieval of video results using Bing's search API.
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).

1.1.3 Extensibility of the framework

As stated in the aims detailed earlier, the extensibility of the framework and its components is a key aspect of its design. In is possible to add, customise and extend all of the components, discussed above, for use in a PuppyIR based application. However, several distinct areas have been selected to be written up for this document, i.e. the process for going about adding new components is detailed. These areas were identified as being the most likely for developers/researchers to wish to extend and are described briefly below.

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, 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). See: *Adding new Search Engine Wrappers* for more details on this.

Query and Result Filters/Modifiers

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

1.1.4 Other features and aspects

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. For details of some of the known Python compatibility issues please consult the *Known issues with the PuppyIR framework* page.

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. See *Building a Standalone PuppyIR Service* for more information.

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)
```

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. See *BaSe Tutorial: Building a PuppyIR/Django Service* for more information.

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.

1.2 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.

1.2.1 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'**.

1.2.2 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

1.2.3 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' (see: *SpellingModifier* for more on this component) 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'** (see *The structure and the PuppyIR repository* for more on the structure of the repository) you will need to install Java:

- Java - this site also contains installation instructions for Java.

1.2.4 Basic Installation

The following sections provide instructions on installing each of the requirements, as detailed in *Basic Requirements*.

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.
>>>
```

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'
```

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.

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'
```

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'
```

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:

```
$ svn export https://puppyir.svn.sourceforge.net/svnroot/puppyir/trunk/framework puppyir
$ cd puppyir
$ python setup.py install # may require 'sudo'
```

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

1.2.5 Installing the Extras

The following sections, provide instructions on installing each of the extra requirements (as detailed in *Extra Requirements*).

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 developers/researchers to take advantage of the many features offered by Django and also to run the prototypes and demonstrators bundled with the framework.

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

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

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'
```

Install Enchant

(Only required if using the 'spelling modifier' - see: *SpellingModifier* for more on this component)

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'
```

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'
```

1.3 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.

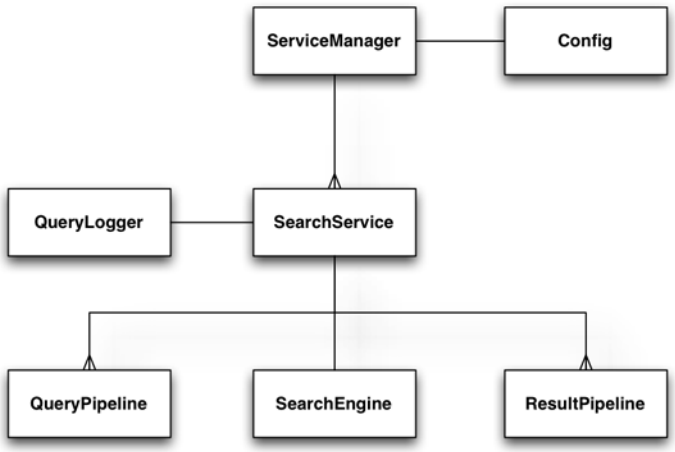

Figure 1.1: *The basic architecture of a PuppyIR application, using the 'Search Service' paradigm.*

1.3.1 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'.

1.3.2 Data flow in the 'Service' paradigm

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

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.

1.3.3 On Filters, Modifiers and Query Logging

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.

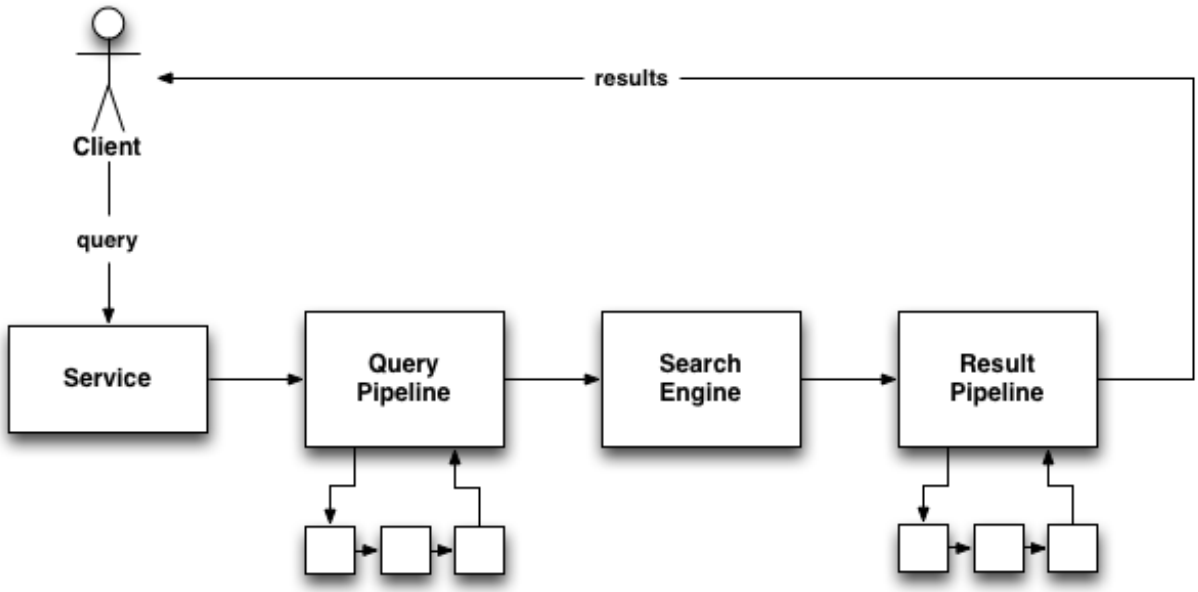

Figure 1.2: *The basic data-flow in the 'Search Service' paradigm.*

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
```

1.3.4 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 (for more see: *Query*).

- The results are in the 'Response' format (for more see: *Response*); 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; for more details, see the links below:

- OpenSearch Query.
- OpenSearch Response.

1.4 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 (*Paradigm 1 - One Pipeline, One Search Engine*), 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:

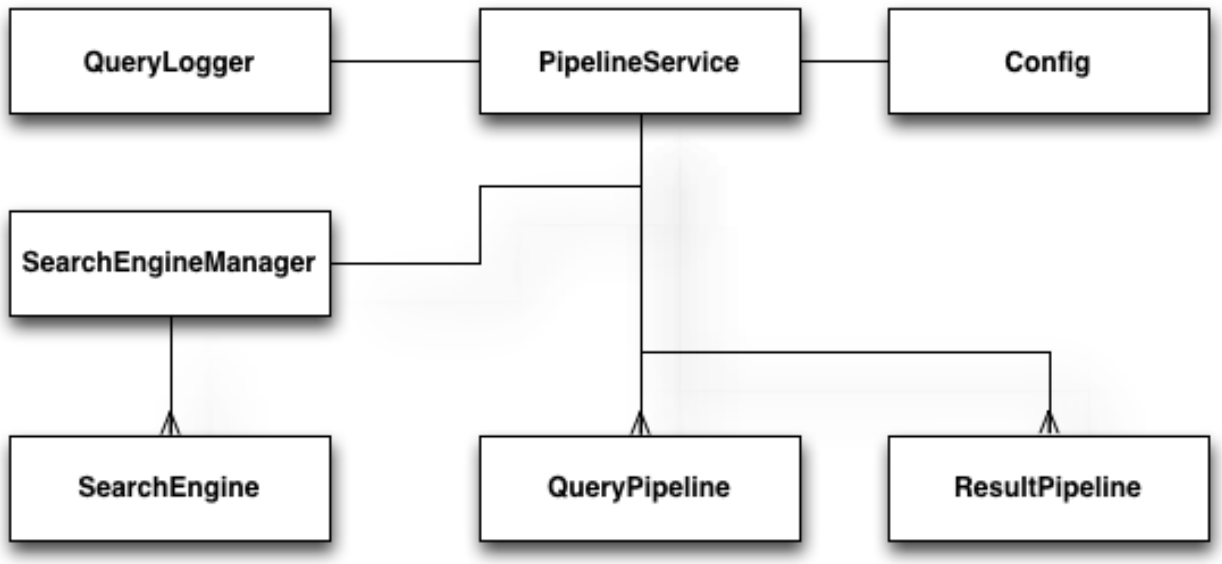

Figure 1.3: *The basic architecture of a PuppyIR application, using the 'Pipeline Service' paradigm.*

1.4.1 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'.

1.4.2 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.

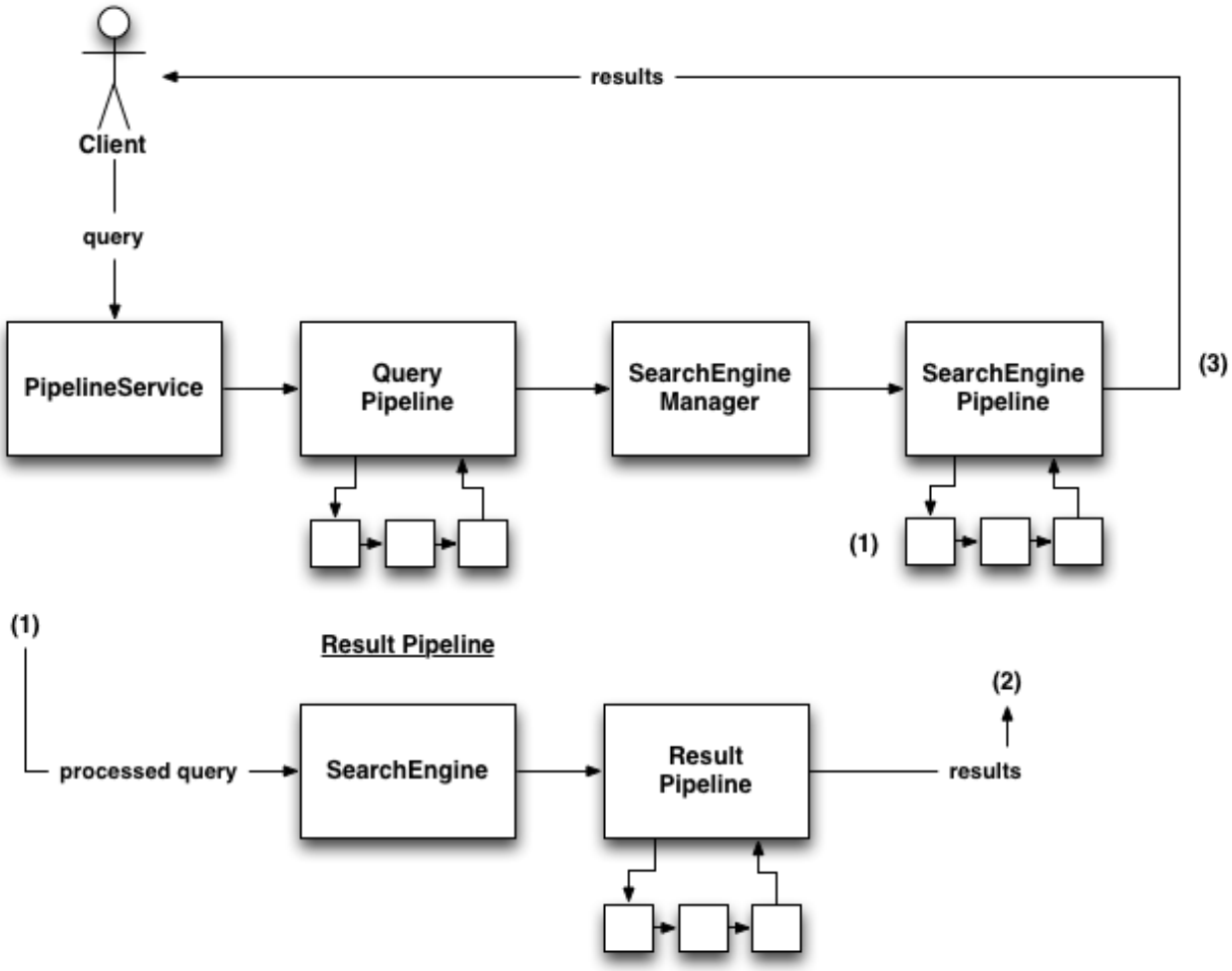

Figure 1.4: *The basic data-flow diagram for the 'Pipeline Service' paradigm.*

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).

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.

1.4.3 On Filters, Modifiers and 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.

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, please consult the *PuppyIR API Reference* page for details of what is available.

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
```

1.4.4 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 (for more see: *Query*).
- 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 (for more see: *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; for more details, see the links below:

- OpenSearch Query.
- OpenSearch Response.

1.4.5 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.

1.4.6 Further Reading

An example of the usage of this paradigm is given in: *Pipeline Tutorial: DeeSe (Detective Search)*.

CHAPTER

TWO

USING THE FRAMEWORK

2.1 Running Prototypes

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

- **aMuSe**: a multimedia search interface for children, allows children to search for videos & images and to be able the explore these results, via the automatic 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 (see: *IfSe Tutorial: Information Foraging Search Application*) for using the PuppyIR framework to create and manage a pipeline.
- **MaSe**: Create Your Own Mash-Up Search Interface: an application created as a tutorial (see: *MaSe Tutorial: Create Your Own Mash-Up Search Interface*) 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.
- **YouSee**: A Video Browsing Application for Young Children - a search service where children can browse moderated video content using a novel interactive paradigm.

2.1.1 Downloading the Prototypes

All the prototypes require Django to be installed to use them. If you do not have Django installed, then please visit the *Requirements and Installation* page and install it before downloading the prototypes.

In addition, IfSe and MaSe also require Whoosh to be installed and so if you want to run these prototypes please visit the installation page, as detailed above, and install Whoosh.

The source code for all these prototype services can then be downloaded with the following command:

```
$ 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 prototype you want to download. If you do this, you will need to amend the paths accordingly to run the prototypes by removing the 'prototypes' part of the path as noted in the **run** sections below. The command would, in this case, be:

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

2.1.2 Using aMuSe: A Multimedia Search Interface for Children

aMuSe allows video and picture results to be retrieved from YouTube and Bing's image search services. 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.

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

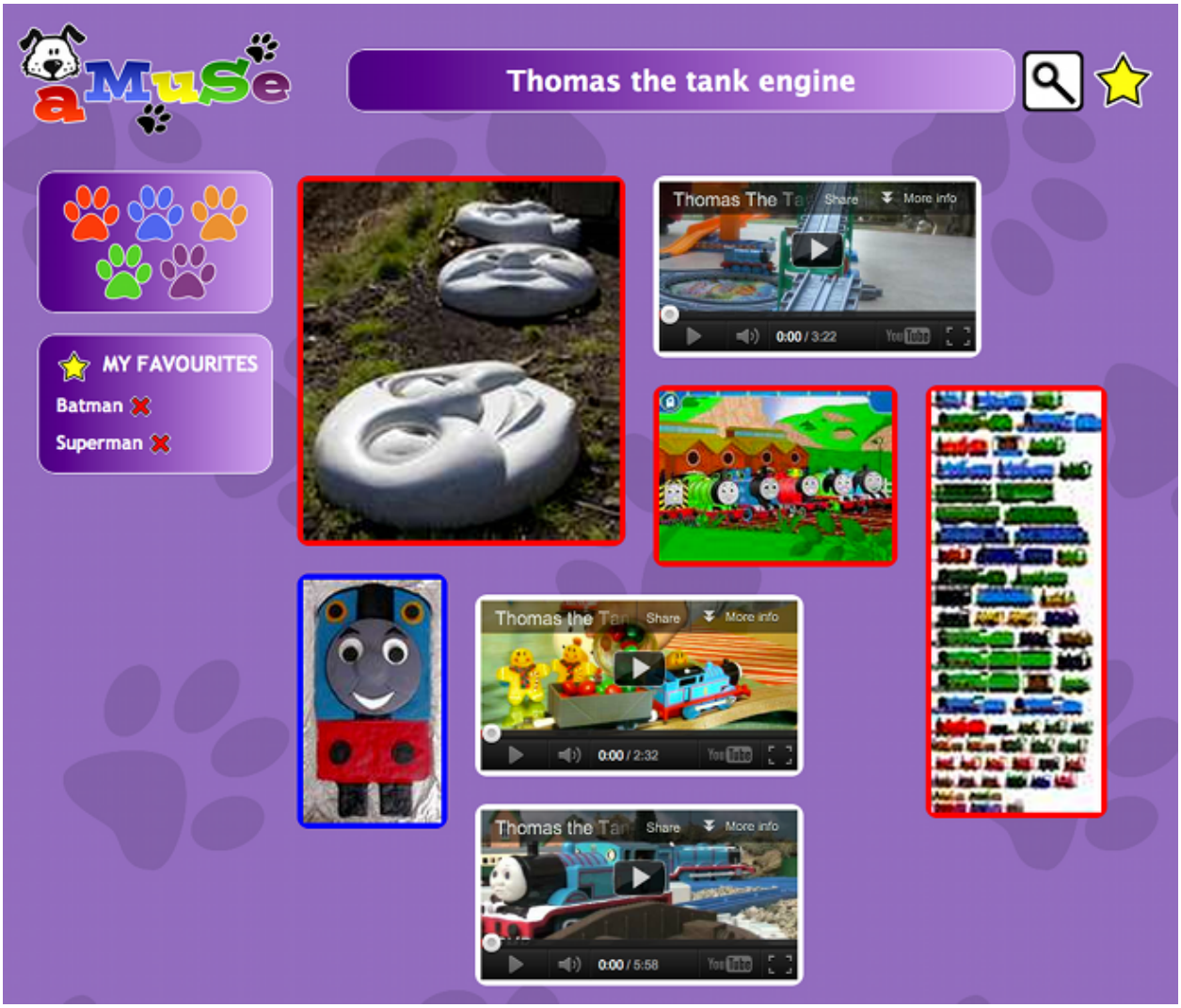

Figure 2.1: *aMuSe showing a video/image collage of 'Thomas the Tank Engine' results.*

The three different methods for generating new queries are:

- **Query Drift**: a new query is generated by randomly removing terms and randomly adding terms to the original query. The terms selected are drawn from the snippets associated to the results on the page or from the snippet itself.
- **Query Specify**: a new query is generated from the snippet text associated with the image by appending the next most informative term to the original query. This generally produces more specific and focused queries that narrow the search results.

- **Query Repeat**: the same query is used again, but this time with a page offset so as to present results which would appear lower down in a ranked list.

Run aMuSe

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

Visit: http://localhost:8000/amuse

2.1.3 Using BaSe: Basic Search

This is a search application with a simple 'google-like' interface. It was created as both a tutorial (see: *BaSe Tutorial: Building a PuppyIR/Django Service*) and as a demo application to illustrate the ease at which interactive applications can be developed using the PuppyIR framework.

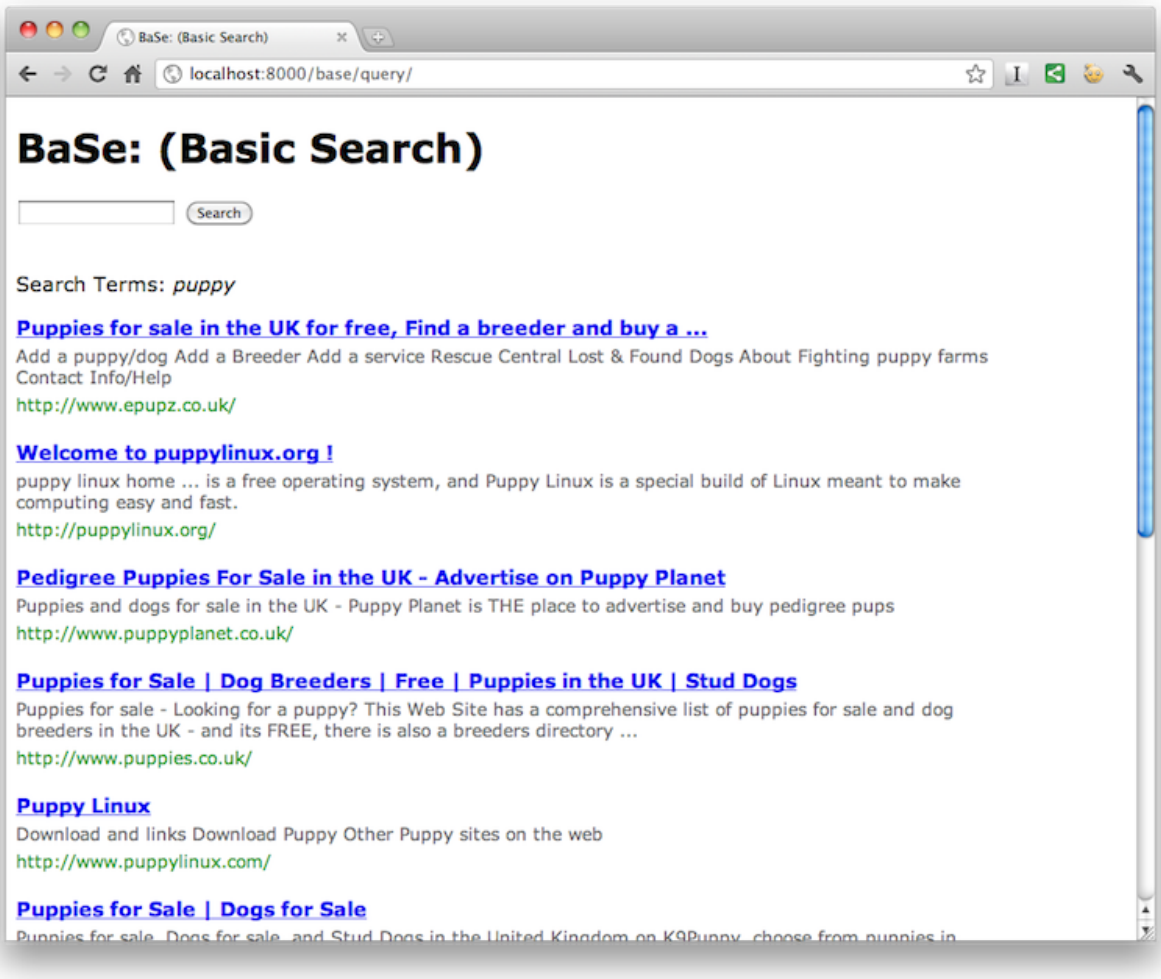

Figure 2.2: *BaSe running on a local machine showing web results for the query 'puppy'.*

Run BaSe

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

Visit: http://localhost:8000/base

2.1.4 Using IfSe: Information Foraging Search

This prototype was created as a tutorial (see: *IfSe Tutorial: Information Foraging Search Application*) designed to teach how to: retrieve results from multiple sources (search engine wrappers), log queries, generate query suggestions and how to create & manage result/query pipelines using the PuppyIR framework.

Figure 2.3: *IfSe running on a local machine showing web results for the query 'puppy'.*

Run IfSe

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

Visit: http://localhost:8000/ifse

2.1.5 Using MaSe: Create Your Own Mash-Up Search Interface

MaSe is an application designed to allow children to create and customise their own search engine - retrieving results from a variety of sources in several different formats (e.g. web results, images, videos). See the MaSe tutorial for more details about the application *MaSe Tutorial: Create Your Own Mash-Up Search Interface*.

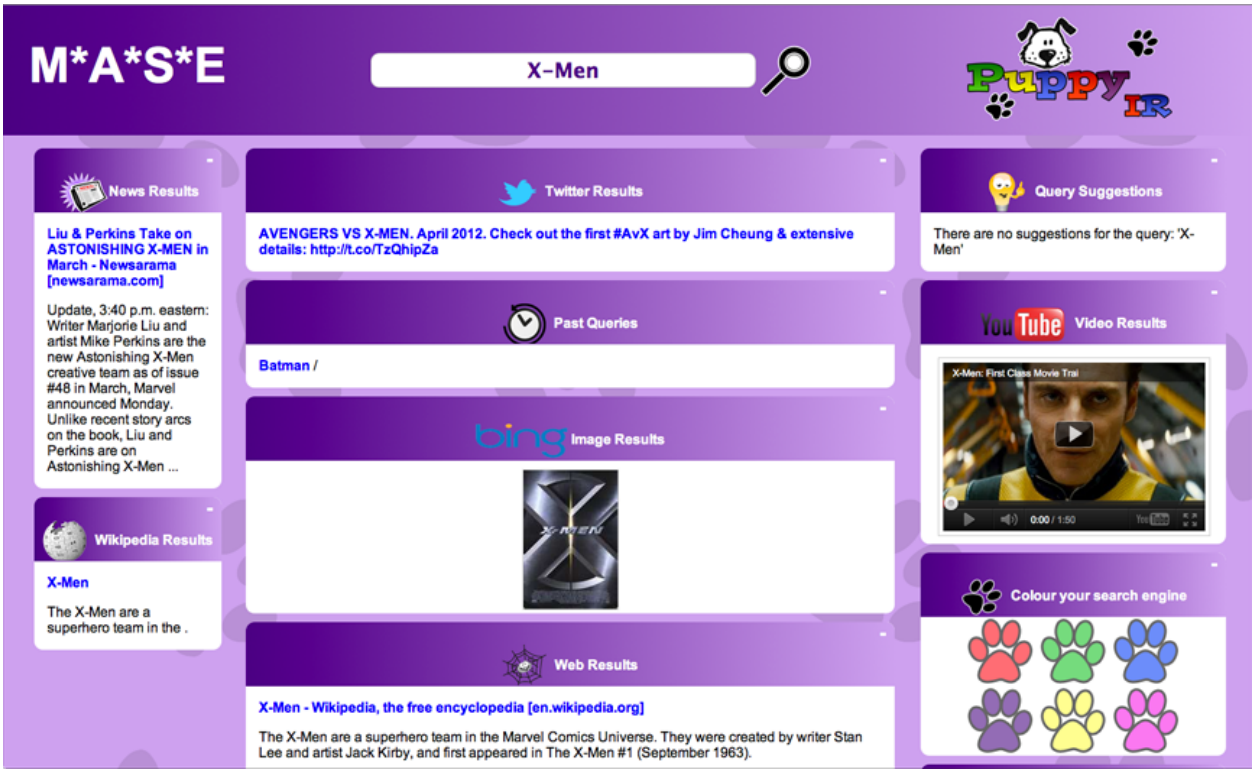

Figure 2.4: *MaSe running on a local machine showing web results for the query 'X-Men'.*

Run MaSe

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

Visit: http://localhost:8000/mase

2.1.6 Using SeSu: Search and Suggest

SeSu is a prototype service that investigates the use of query suggestions and the effectiveness of the framework's suitability filter. This filter looks at individual results and evaluates their suitability for children; it then decides

whether to accept or reject the result, based on a defined cutoff minimum score they should receive.

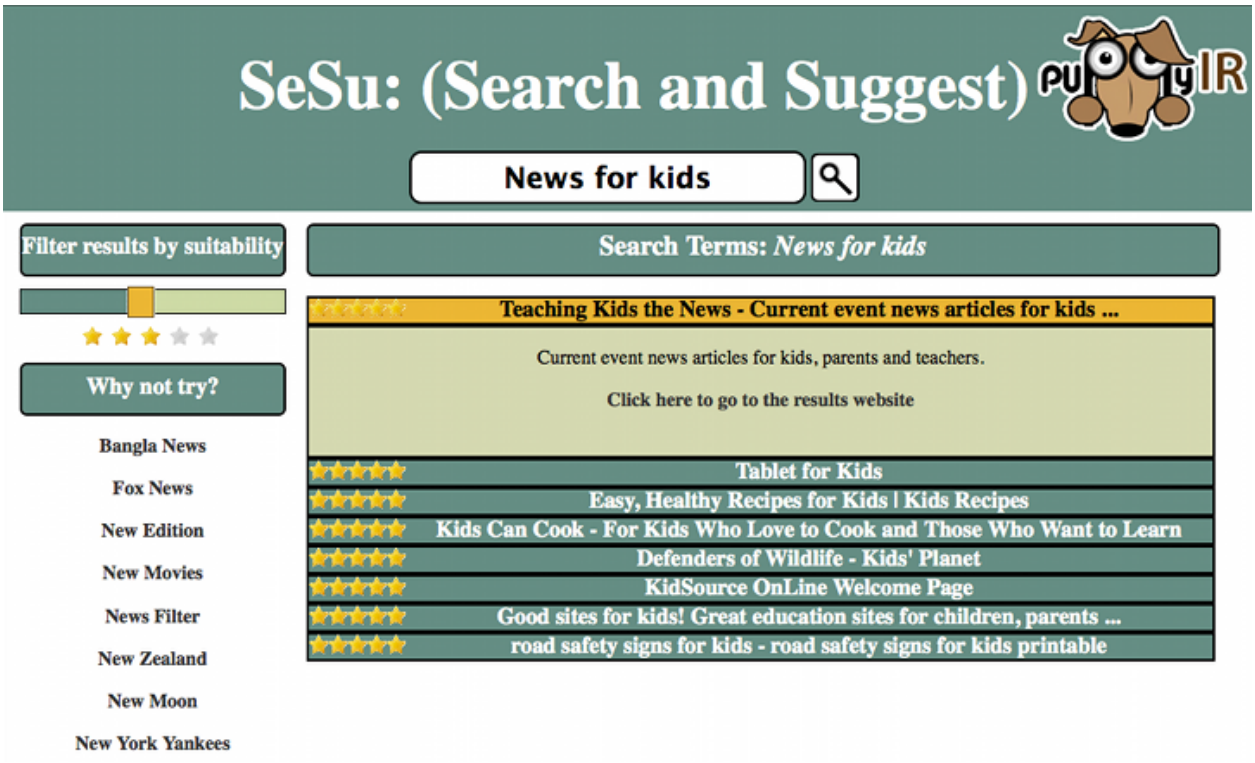

Figure 2.5: *SeSu showing results, with their suitability rating, for a query about the news.*

Run SeSu

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

Visit: http://localhost:8000/sesu

2.1.7 Using YouSee: A Video Browsing Application for Young Children

YouSee is a web application designed to allow young children to browse moderated video content (the videos are moderated and managed by their parents using the admin interface). The application aims to avoids the problems associated with text query formulation by young children by providing a novel interaction paradigm based on a globe of videos for children to explore. Conceptually the globe can be thought of as a series of carousels containing videos where the interaction paradigm allows two forms of browsing/scrolling:

1. in carousel to access similar and related content
2. between carousels to access different content.

Run YouSee

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

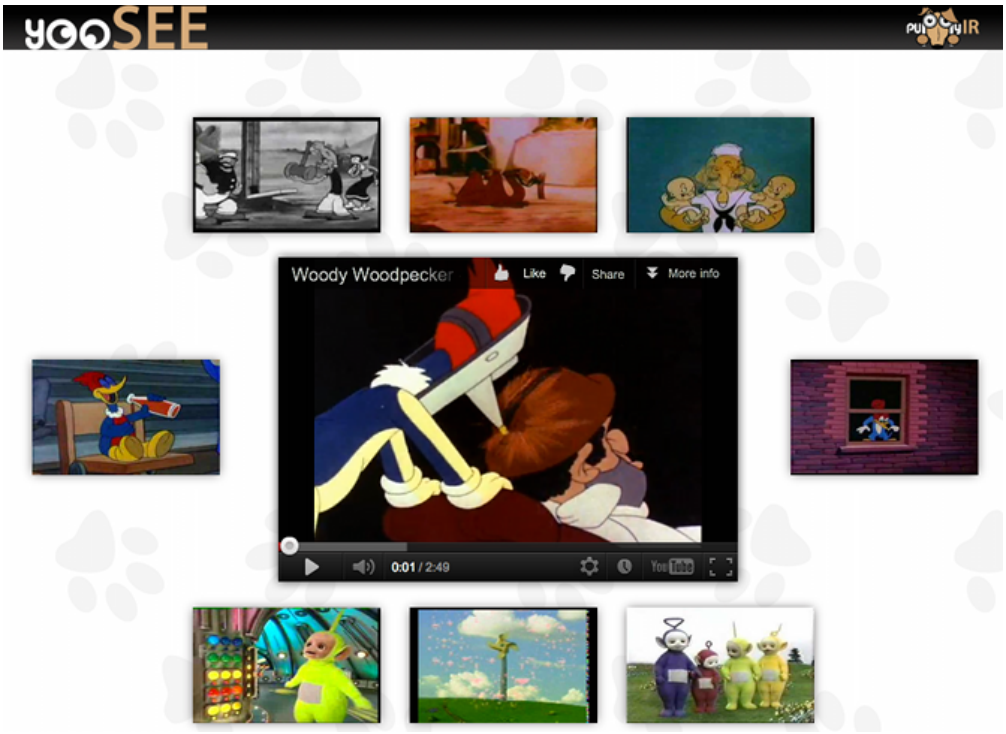

Figure 2.6: *Browsing between carousels using YouSee*

Visit: http://localhost:8000/yousee

2.2 Building a Standalone PuppyIR Service

The PuppyIR framework can, in addition to being used in combination with frameworks like Django (this is detailed later in the *BaSe Tutorial: Building a PuppyIR/Django Service*), be used to build a standalone services with no graphical 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.

This section assumes that you have read the section of the search service paradigm (if not, read it now before going any further: *Paradigm 1 - One Pipeline, One Search Engine*) and are familiar with its various components.

The following steps will create and configure a new service, consisting of: a search engine, a query logger, a query pipeline, a result pipeline and query suggestions. The code comments note where new lines of code are and what they do.

2.2.1 Create and configure a ServiceManager

Create a new python script called *service.py* and add the following lines of code to it to create a service manager:

```
from puppy.service import ServiceManager

config = {} # See note below on Proxy Servers

# Create the ServiceManager
sm = ServiceManager(config)
```

N.B. if you require this standalone service to go through a proxy server, amend the config line of code to:

```
config = {"proxyhost": "http://your-proxy-server-address"}
```

2.2.2 Create a SearchService

Next, we will create a search service to add to our service manager ready to retrieve results. Amend your code so it matches the following:

```
# We now need to import SearchService as well
from puppy.service import ServiceManager, SearchService

config = {}

sm = ServiceManager(config)

# Create SearchService and give it a name
ss = SearchService(sm, "bing_web")

# Add our new SearchService to ServiceManager
sm.add_search_service(ss)
```

2.2.3 Add a SearchEngine

Of course, our search service has not yet been linked to a search engine wrapper, let's link it to Bing by amending the code like so:

```
from puppy.service import ServiceManager, SearchService

# Import PuppyIR's Bing search engine wrapper
from puppy.search.engine import Bing

config = {}

sm = ServiceManager(config)
ss = SearchService(sm, "bing_web")
sm.add_search_service(ss)

# Set our SearchService to use the Bing wrapper
ss.search_engine = Bing(ss)
```

2.2.4 Perform a Search

At this stage, we can now use the service we have created to search using Bing and retrieve results. Let's add some code to handle this and output the results to console:

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

# Import PuppyIR's Query and Response models
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)

# Construct a query object for the query term puppy
query = Query("puppy")

# Retrieve the results from our SearchService (.entries are the results in a Response)
results = sm.search_services['bing_web'].search(query).entries

# Go through each result and output the title, summary and link they contain
for result in results:
  print result['title']
  print result['summary']
  print result['link']
  print '\n'
```

2.2.5 Enable the QueryLogger

It may be useful to start logging queries to file so we can keep track of our query history:

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

# Import PuppyIR's QueryLogger
from puppy.logging import QueryLogger

# Add a log_dir and set the path to it in config
config = {"log_dir": "/path/to/log/directory"}

sm = ServiceManager(config)
ss = SearchService(sm, "bing_web")
sm.add_search_service(ss)
ss.search_engine = Bing(ss)

# Assign a QueryLogger to our 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']
  print '\n'
```

2.2.6 Adding a QueryModifier and a ResultFilter

Now that we have an application that retrieves results up and running let's tailor it to suit children. First, we'll add a query modifier to append 'for kids' to all our queries and second, a suitability result filter to remove unsuitable results (for children):

```
from puppy.service import ServiceManager, SearchService
from puppy.search.engine import Bing
from puppy.model import Query, Response
from puppy.logging import QueryLogger

# Import the modifier and filter mentioned above
from puppy.query.modifier import TermExpansionModifier
from puppy.result.filter import SuitabilityFilter

config = {"log_dir": "/path/to/log/directory"}

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 a TermExpansionModifier to SearchService
ss.add_query_modifier(TermExpansionModifier(terms='for+kids'))

# Add a SuitabilityFilter to SearchService - see note below on threshold
ss.add_result_filter(SuitabilityFilter(threshold=0.5))

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 out the score each result (that was accepted) received
  print result['suitability']

  print '\n'
```

N.B. this filter works out a score for each result, which ranges from: 0.0, not suitable for children to 1.0, very suitable for children. The threshold defines the cutoff score for whether a result is accepted or rejected (i.e. only accept results with a score greater than 0.5). Try playing about with different settings for the threshold and investigate which results don't make the cut.

2.2.7 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). Amend your code to match the following code and try out a few queries to see what suggestions you receive:

```
from puppy.service import ServiceManager, SearchService
from puppy.search.engine import Bing
from puppy.model import Query, Response
from puppy.logging import QueryLogger
from puppy.query.modifier import TermExpansionModifier
from puppy.result.filter import SuitabilityFilter

config = {"log_dir": "/path/to/log/directory"}
```

```
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)

ss.add_query_modifier(TermExpansionModifier(terms='for+kids'))

ss.add_result_filter(SuitabilityFilter(threshold=0.0))

# Create a new SearchService for our query suggestions service
suggestions_service = SearchService(sm, "suggestion_search")

# Set our new SearchService to use the Bing wrapper with RelatedSearch
suggestions_service.search_engine = Bing(suggestions_service, source = "RelatedSearch")

# Add our new SearchService to our ServiceManager
sm.add_search_service(suggestions_service)

query = Query("puppy")
results = sm.search_services['bing_web'].search(query).entries

# Retrieve our query suggestions
suggestions = sm.search_services['suggestion_search'].search(query).entries

for result in results.entries:
  print result['title']
  print result['summary']
  print result['link']
  print result['suitability']
  print '\n'

# Go through and print out our query suggestions to console
for result in suggestions:
  # The title is the query suggestion, i.e. for Batman a suggestion could be: Batman Begins
  print result['title']
```

2.3 Exception Handling in PuppyIR

The PuppyIR framework provides a basic set of exceptions that handle errors that can occur in 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. This section details the handling of these exceptions and provides some examples.

2.3.1 Exception handling in the Query Pipeline

The following exceptions are available in this area of the framework:

- **Query Rejection Error**: used 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**: used when a filter operationally failed and the filter's function cannot be realised. Callers should respond to this as if a query rejection decision cannot be made.

- **Query Modifier Error**: used when a modifier operationally failed and the modifier's function cannot be realised. Callers should respond to this as if the query has not been modified as per the design of the developer.

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
```

2.3.2 Exception handling for searching within an application

The following exceptions are available at this stage:

- **Search Engine Error**: used 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 use exception that deals with any problems that might occur during the operation of a search engine wrapper.
- **API Key Error**: used only in a search engine wrapper that requires 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 is given below:

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

2.3.3 Exception handling in a search engine wrapper

The following two examples detail how to implement the exceptions detailed above, in the search engine wrapper itself. I.e. if you are extending this area of the framework (see: *Adding new Search Engine Wrappers* for more details on adding a new search engine wrapper) or simple want to look at the code to understand what it's doing.

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:

1. A urllib2 error and add in extra parameters for the error message.
2. A type error for some local variables.
3. A general catch-all error for anything unforeseen (this enables the **SearchEngineError** to be used in an application as a general catch all exception).

```
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, url = url)

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

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

2.3.4 Exception handling in the Result Pipeline

The following exceptions are available at this stage:

- **Result Filter Error**: used when a filter operationally failed and the filter's function cannot be realised. Callers should respond to this as if a rejection decision cannot be made for the results of a query.
- **Result Modifier Error**: used when a modifier operationally failed and the modifier's function cannot be realised. Callers should respond to this as if the results have not been modified as per the design of the developer.

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

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

2.3.5 Generic Error Handling

Error handling within a wrapper, filter or modifier is the responsibility of the developer that created the component. However, the PuppyIR framework does provide some basic generic exception handling. All of the components listed

above contain a variable called *handleException* which defines if they should gracefully fail (e.g. if a query modifier fails then just return the un-modified query and continue) if there's a problem, or raise the exceptions listed above. You can set this for each individual component depending its importance, i.e. if our profanity filter fails, do we want to not return any results or continue on - despite the possibility of profanity in a query.

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.

To create a test, where <module> is the name of the Python file the test is for, use the following commands:

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

To run all the tests, currently defined in the test suite, use the following commands:

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

2.4.1 Create

The **Create** command generates a skeleton python script. This script is placed at a location in the test hierarchy that mirrors where the component being tested is in the framework's hierarchy.

For example, if we wanted to create a test script for our query filter, cool_filter we should use the following commands:

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

Our test script would be created in: test/puppy/query/filter/cool_filter.py (relative to our framework directory) - 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.

2.4.2 Run

The **Run** command searches for all the current test cases and runs each of them in turn. Any issues are reported at the end of this process; nothing is outputted if a test succeeds, so if you run this command and nothing is outputted there are no problems.

If you are using a proxy server, there are two options:

1. Either use the in-built proxy system using a ServiceManager or PipelineService (via the config variable) in your tests.

2. Write a work-around for your tests, or they will fail due to proxy errors (unless, of course, you are testing a component that does not need to go through the proxy server).

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()
```

CHAPTER

THREE

TUTORIALS

3.1 BaSe Tutorial: Building a PuppyIR/Django Service

The BaSe (Basic Search Engine) tutorial details how to create an application using the Django web application framework and the PuppyIR framework. Before starting this tutorial, please ensure that you have followed the instructions on *Requirements and Installation* for installing the framework (and its dependencies) 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.

3.1.1 Creating a Django project and application

First, browse to the directory you want to store BaSe in and run the following commands to create a Django project for our application:

```
$ 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 success page should be displayed congratulating you on creating your first Django project.

Now, we will create an application within the BaSe project called WeSe (Web Search). 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 the WeSe application:

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

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

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

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

Add a new directory called *template* in the BaSe folder. 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" "http://www.w3.org/TR/html4/strict.dtd">
    <html>
      <head>
        <title>WeSe (Web Search) - a BaSe application</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      </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 include 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, we can then use this to refer to the template directory's path relative to this variable (no need to hard code the absolute path). Amend the template directory's code (in *settings.py*) so it looks like this:

```
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 this:

```
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
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('index.html', context)
```

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

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 configure them. 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 '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)).entries
    context = RequestContext(request)
    results_dict = {'query': user_query, 'results': results}
    return render_to_response('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!

3.2 IfSe Tutorial: Information Foraging Search Application

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 also require Whoosh to be installed). If you have not installed the PuppyIR framework and/or Whoosh please go to *Requirements and Installation* and get everything set up.

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 interactive 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 to complete the tutorial are provided below, along with comments and a step-by-step guide on how to go about writing the code.

Downloading the Source Code for the Tutorial

First, download the latest release of the tutorial from the PuppyIR repository with the following command:

```
$ 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 append ' ifse-tutorial' to the above command.

Run IfSe

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

Now, visit: http://localhost:8000/ifse which should bring up interface shown below.

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 allows you to search the web, it doesn't record anything.

3.2.2 Logging Queries

Let's assume that we'd like to keep track of all the queries that users submit, this is so that we can do query log analysis later on to evaluate IfSe and how its users are using it.

There are a number of ways to do this, but let's do it the simplest 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 (see *PuppyIR API Reference* for more information).

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

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

This is a flat file based query logger. To tell PuppyIR where to store the log, we need to add a *log_dir* entry to the config dictionary, do that now:

```
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://localhost:8000//ifse

Type in a few queries.

Open the `ifse/query_logs` directory and you should see a file 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!

3.2.3 Modifying Queries

One of the PuppyIR project's aims is to create various child friendly search services. So lets add some new components to tailor IfSe for children, first lets add a QueryFilter to stop naughty query terms being submitted to IfSe. To do this, we can use the `BlackListFiter` component that is part of the PuppyIR framework. Add the following line of code to import it:

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

Figure 3.1: *IfSe running on a local machine.*

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 `BlackListFiter` 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.

3.2.4 Adding Another Search Service

Instead of just returning web results, we might want to all add in other kinds of results. The PuppyIR framework also contains various search engine wrappers to API's other than Bing, such as: Twitter, Yahoo, etc (for more details about search engine wrappers see the *PuppyIR API Reference*).

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 lines of code to import the Twitter search engine wrapper and create a Twitter search service:

```
from puppy.search.engine import Twitter

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 it should return two panes of results: first, the web results and second, the twitter results.

Wow! How cool is that?

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 to include two new components: a `QueryFilter`, that records and indexes queries submitted to the service, and a third `SearchService`, that recommends new 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 wrapper 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` and provide it with the full path name to the index directory. It then needs to be added to the `search_service` that we wish to log queries for, let's add it to 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)
```

Next, we want to provide the query suggestions, so we need to create a SearchService for query suggestions and then set it to use the `WhooshQueryEngine` wrapper, this 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)
```

Okay, so let's start entering a few queries. N.B. you might have to enter a few queries before you start to see recommendations appearing.

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.

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 the `templates/ifse/` directory and the corresponding style sheets 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 between `perplex`, `combination`, `passageway`, `twirling` or download any other CSS styles from http://freecsstemplates.org and try them out (by adding the files to the `site_media/css/` directory).

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 also really easy to develop your own unique components to further customise your search service to suit its, and your users, individual needs. To develop your own components, check out *Extending the Query Pipeline*, *Extending the Result Pipeline* and *Adding new Search Engine Wrappers* Chapters for more details.

3.3 MaSe Tutorial: Create Your Own Mash-Up Search Interface

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 go to *Requirements and Installation* and get everything set up.

The MaSe tutorial is designed to show the PuppyIR framework can be used to create an interactive customisable web application, quickly, using the Django web application 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 for you.

You will also need a Bing Azure API key, you can sign up at https://datamarket.azure.com/dataset/5BA839F1-12CE-4CCE-BF57-A49D98D29A44 and obtain a key for for free.

Downloading the Source Code for the Tutorial

The first step is 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 mase-tuto
```

Run 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):

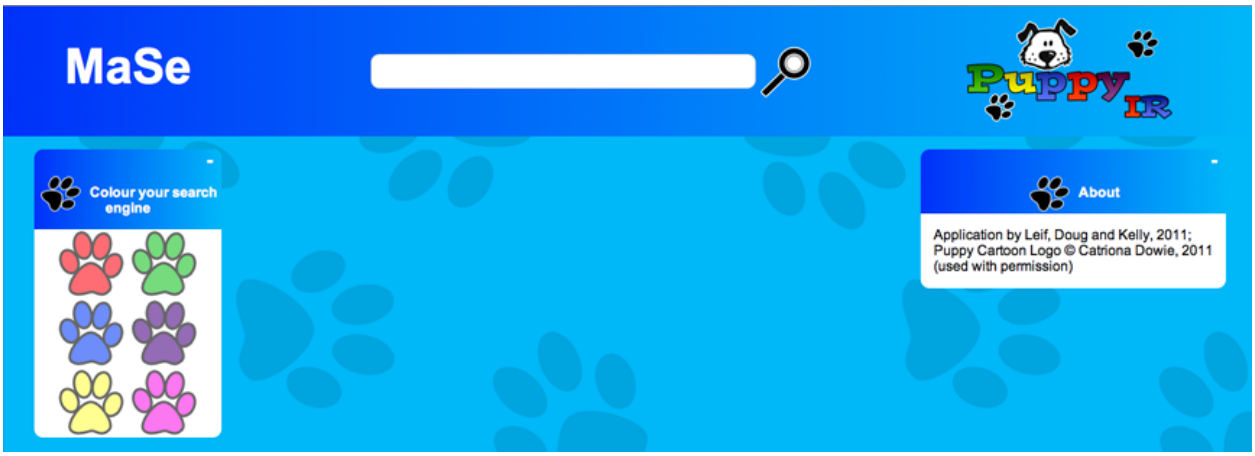

Figure 3.2: *MaSe running on a local machine.*

To search for results either press enter/return in the search box or click on the magnifying glass. You will notice that no results are returned - this is because we have to construct a PuppyIR service and configure it to include a search engine.

You can customise several aspects of the MaSe search interface, as detailed below:

1. 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.
2. Clicking on the paw images in the *Colour your search engine* box will change the colour theme.

3. You can also move the result boxes around on the screen (more on this in the next section).
4. 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 now 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.

3.3.2 Adding our first services

Since, we don't have any services added yet we wont get no results when searching. Let's fix that now by adding our first search 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 of code):

```
# 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 = BingV3(web_search_service, source='Web')

# add SearchService to our ServiceManager (this handles all the search services MaSe contains)
service.add_search_service(web_search_service)
```

Now that you have added the service you'll need to make sure you have entered your Bing Api key into the config:

```
config = {
  ....
  ....
  "bing_api_key": "<--PUT YOUR BING API KEY HERE--->",
}
```

You may also need to set the proxy in the config file. Ask your teacher if you need to add a proxy and add it to the config aswell.

Now, save this file and refresh your browser and search for something (try the query "puppies"). You should be presented with results, for your query, in a format similar to what is shown below:

Now, lets limit the number of web results to only three, this is done by changing the line of code shown below to:

```
# Set the resultsPerPage parameter to 3; this limits the results the service will return
web_search_service.search_engine = BingV3(web_search_service, source='Web', resultsPerPage = 5)
```

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 = BingV3(image_service, source='Image', resultsPerPage = 5)

# add SearchService to our ServiceManager
service.add_search_service(image_service)
```

Go ahead and search for something, you should now see both image and web results displayed. 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:

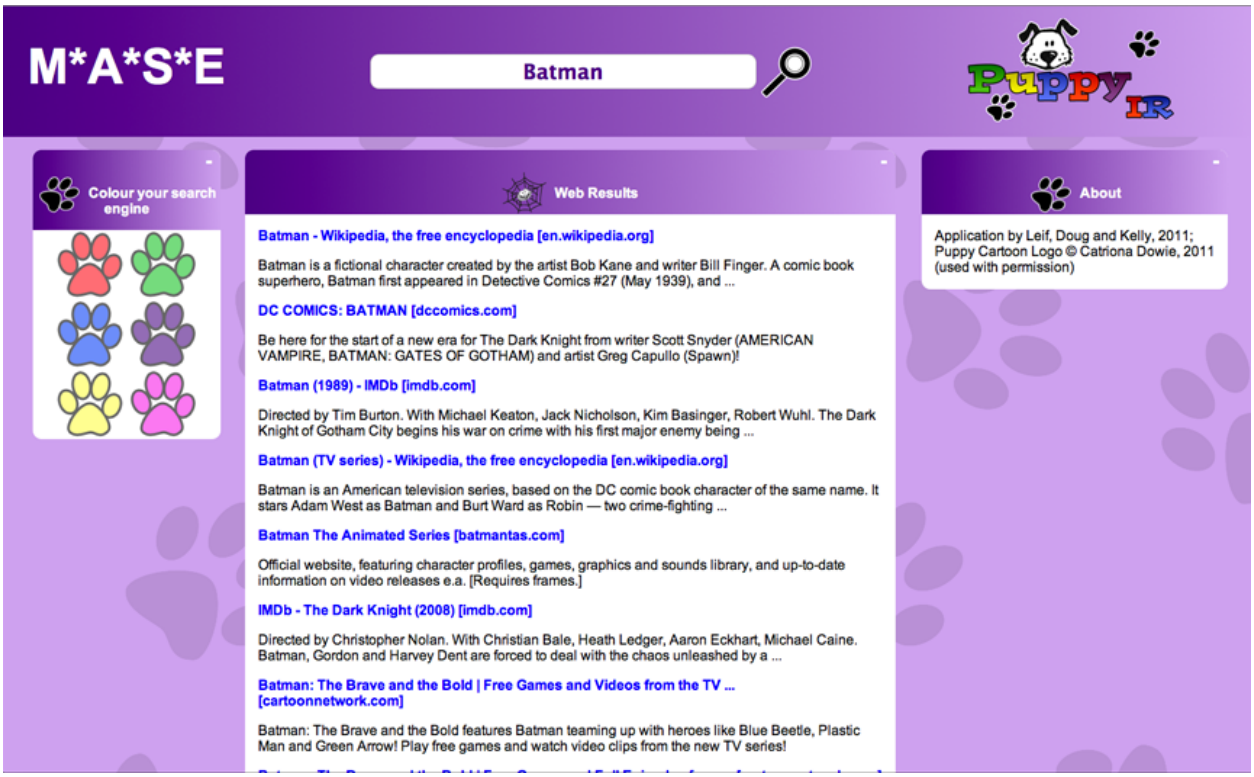

Figure 3.3: *Our now customised MaSe (custom title and new colour scheme) showing web results.*

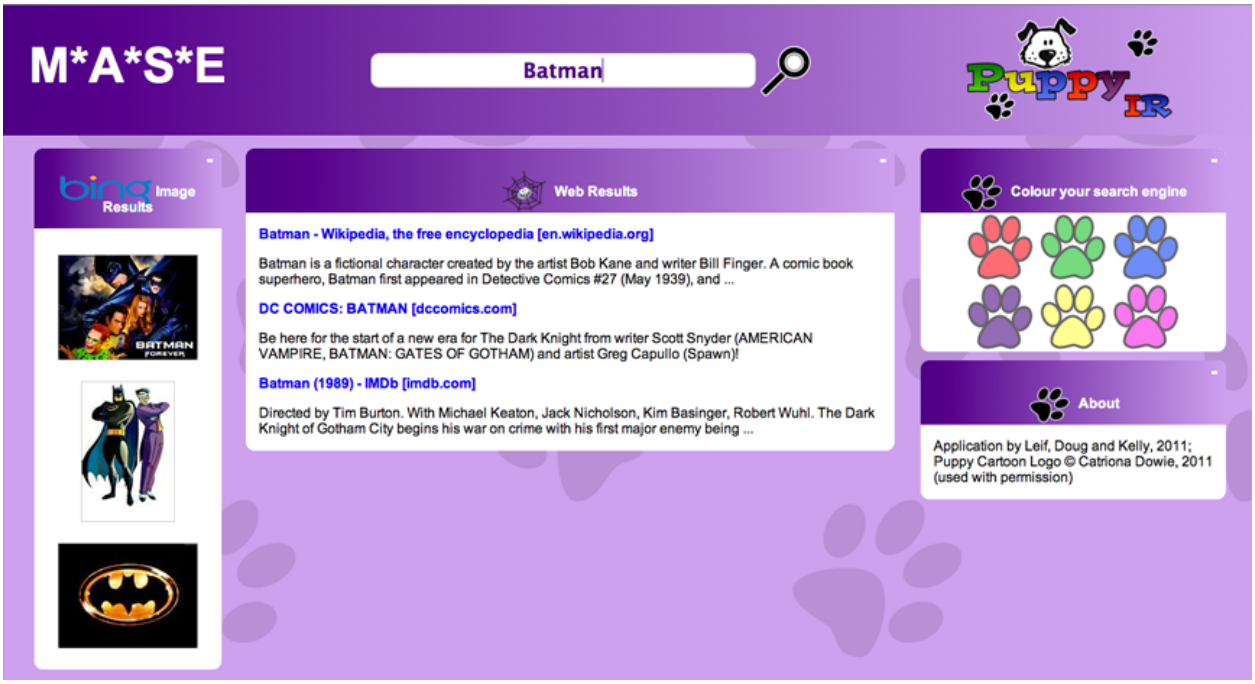

Figure 3.4: *Re-arranging 'Web' and 'Image' results in MaSe.*

3.3.3 Extending MaSe with query logging and suggestions

Now let's add a query logger to record the queries that users submit. Add the code below, just after where we created the web and image search services:

```
# create a file based QUERY LOGGER
web_search_service.query_logger = QueryLogger(web_search_service, log_mode=0)
```

Now save the file again, and try a few queries: "puppies", "kittens", "cats and dogs"...

This simple query logging component simply saves the queries that were entered into the a file called "web_search_query.log" and it is located in the directory mase-tutorial/mase/query_logs. Take a look at the query log.

This query logger is pretty simple, but what if we want to provide query suggestions. In this case, it would be preferable to save the queries that are entered into a query index, which we can search later on. To do this add the following lines of code:

```
# 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)
```

By adding this code, we now store all the queries in an index, which is housed in the directory: mase-tutorial/mase/query_logs/index

To provide suggestions to the user we need to add another search service, one which submits queries to this query index. To enable this feature, add the following lines of code:

```
# 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. Please note, you may need to enter a few queries before any suggestions start appearing, as some queries need to be recorded before they can be recommended as suggestions.

3.3.4 Filtering and the Pipelining

Now that we've can retrieve results from three different sources (Web, Images, and Queries) we can begin to further customize our search services. By adding in the WhooshQueryLogger we added our first filter to the query pipeline.

Let's now add a query modifier to the query pipeline to stop users from submitting certain "blacklisted" terms. To do this after the creation of the *web_search_service*, add the following lines of code to add a **BlackListModifier** to remove inappropriate terms from queries:

```
# Create a blacklist modifier to block queries containing the terms below
query_black_list = BlackListModifier(terms = "bad worse nasty filthy")
```

Figure 3.5: *MaSe showing our now limited results for each service and query suggestions.*

```
# Add our blacklist modifier to the web search service
web_search_service.add_query_modifier(query_black_list)
```

Save the file and type in a few queries, like "bad puppy" or "bad kitty". What happens to the results that you see?

You should notice that if you type in "bad puppy", you still get results for "bad puppy" from the image service. This is because we didn't add a **BlackListModifier** to our *image_service*. Do that now and make sure nothing nasty gets through.

Okay, so now try a few more queries similar to the ones you have already typed, "puppy", "nasty puppy", "puppy", etc. Do you notice anything strange about the query suggestions? Do they recommend queries that include the terms, "bad" or "nasty"?

This is not desireable, because we don't want to recommend inappropriate queries. The reason why this occurs is because we added the **BlackListModifier** after the **WhooshQueryLogger**, which means we first record the query before modifying it. Then when we querying the **WhooshQueryEngine** it retrieves these inappropriate queries.

Fix the order of the modifers and filters to avoid this problem. You may need to delete the query index in mase-tutorial/mase/query_logs/index (i.e. delete all files _MAIN_*.* in that directory).

Note that **QueryLogger** we attached the web search service, by default logs all queries without the filters and modifiers applied. i.e. it is the raw query log. To log the processed queries, you need to set:

```
web_search_service.postLogging = True
```

Further note that **QueryModifiers** are executed before **QueryFilters**.

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.

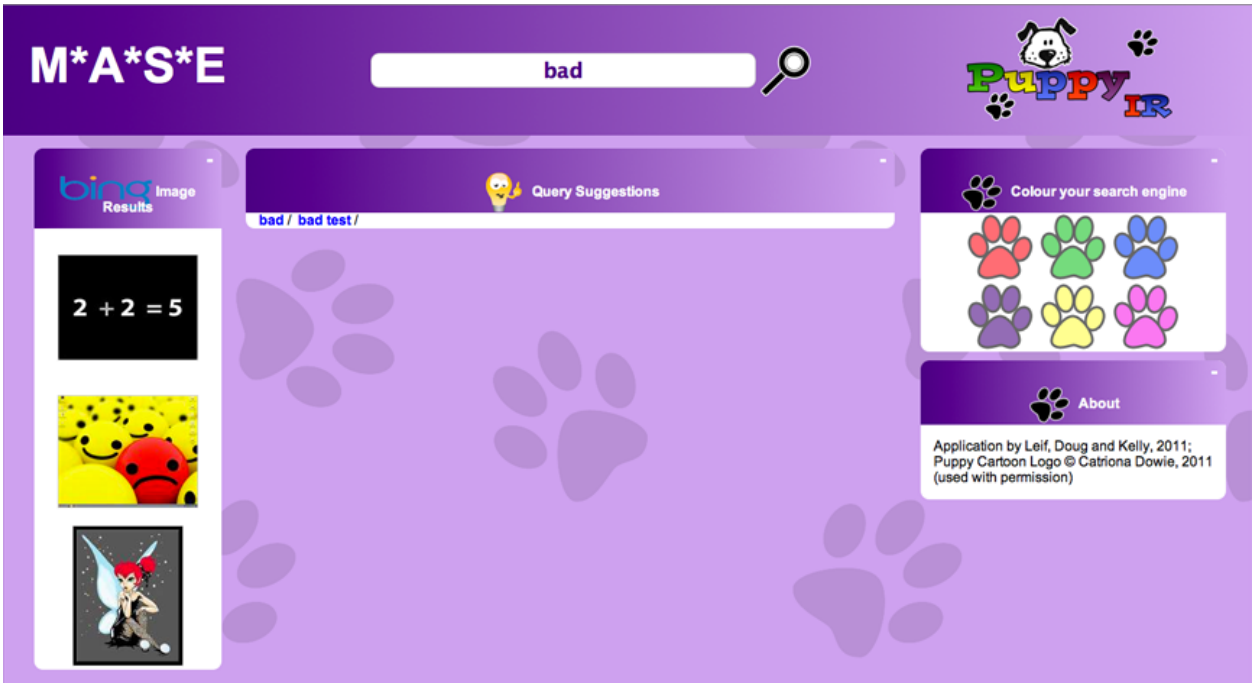

Figure 3.6: *MaSe making bad suggestions and still showing image results; as in this case the filter was not added to image search*

Other Services

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

The table below details some other options for other search services that can be added to MaSe, see the code for *web_search_service* and adapt it using the details provided below:

Result Source	Service Name	Class Name	Extra parameters
Wikipedia	*wiki_search*	**Wikipedia**	
Bing News	*news_search*	**BingV3**	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.

3.4 Pipeline Tutorial: DeeSe (Detective Search)

3.4.1 Getting Started

If you have not installed the PuppyIR framework and/or Django, please go to *Requirements and Installation* to get everything set up. Also, before starting this tutorial, it is recommended that you read the background page on the pipeline paradigm (*Paradigm 2 - One Pipeline, Many Search Engines*) as this provides a conceptual description of the various components and how they work together.

The first step is to checkout the latest version of 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.

3.4.2 DeeSee 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.

3.4.3 Creating our Pipeline Service

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

```
# 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)
```

We can now search for Bing News results using the DeeSe's search box. However, there is no filtering yet implemented... we should start creating our pipeline.

3.4.4 Setting up our query pipeline

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

3.4.5 But what about the results?

The other required condition was that the results returned should not contain the suspects names. To do this we need to create a result pipeline to process the results. Let's add a **BlackListModifier**, 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, lets 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 and 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.

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 for the search engines, i.e. they have Web, Wikipedia, News as their titles. 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 (see: *Paradigm 2 - One Pipeline, Many Search Engines*) can be processed by an application.

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 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]

CHAPTER

FOUR

EXTENDING THE PUPPYIR FRAMEWORK

4.1 Extending the Query Pipeline

This section details adding new Query Filters and Query Modifiers.

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

4.1.1 The Query Operator base class

Both filters and modifiers extend the base class QueryOperator class, which is included below for reference purposes:

```
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). It is the base class for both the QueryFilter and QueryModifier classes, which are detailed in the following sections.

4.1.2 Creating new Query Filters

All Query Filters must extend the base class QueryFilter in order to be compatible with the other PuppyIR components. Like with the QueryOperator class it is included for reference purposes below:

```
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.

Example Query Filter

For example, a **BlackListFilter** that rejects queries if they contain blacklisted words could be written as follows:

```
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)
```

Note, in the above example, what needs to be done to conform to the QueryFilter standard. Beyond this, what a new filter does is up to you - the developer.

4.1.3 Creating new Query Modifiers

All Query Modifiers must extend the base class QueryModifier in order to be compatible with the other PuppyIR components. It is included for reference purposes below:

```
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.

Example Query Modifier

For example, a **TermExpansionModifier** that appends extra terms onto a query for example adding "for kids" to each query could be written as follows:

```
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
```

Note, in the above example, what needs to be done to conform to the QueryModifier standard. Beyond this, what a new modifier does is up to you - the developer.

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', this parameter is used to indicate the precedence of the filter or modifier in question.

4.2.1 The Orderable base class

Both filters and modifiers extend the base class Orderable, which is included below for reference purposes:

```
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.

4.2.2 Creating new Result Filters

All Result Filters must extend the base class ResultFilter, in order to be compatible with the other PuppyIR components. Like with the Orderable class it is included for reference purposes below:

```
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.

Example Result Filter

For example, a **ProfanityFilter** that rejects results if their title does not pass the WDYL, or 'What Do You Love', service's test (this is a Google web service that checks for the presence of naughty words):

```
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
```

Note, in the above example, what needs to be done to conform to the ResultFilter standard. Beyond this, what a new filter does is up to you - the developer.

4.2.3 Creating new Result Modifiers

All Result Modifiers must extend the base class ResultModifier in order to be compatible with the other PuppyIR components. It is included for reference purposes below:

```
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.

Example Result Modifier

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
```

Note, in the above example, what needs to be done to conform to the ResultModifier standard. Beyond this, what a new modifier does is up to you - the developer.

4.3 Adding new Search Engine Wrappers

This section details adding new search engine wrappers.

4.3.1 Creating new Search Engine Wrappers

Every search engine wrapper must extend the base class SearchEngine. This base class defines the standard attributes common to all search engine wrappers and also provides the facility to use a search engine wrapper using a proxy server, if this is required. The key aspect, for new search engine wrappers, is that the search method must be overwritten in them (to handle the retrieving of and processing of results from the external web service the wrapper is for).

The SearchEngine base class is included below for reference purposes:

```
# -*- coding: utf8 -*-

import urllib2
```

```
class SearchEngine(object):
  """Abstract search engine interface."""

  def __init__(self, service, **args):
    """
    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()

    # Prints invalid parameters received for the Search Engine
    for parameter in args:
        print "'{0}' received invalid parameter called: '{1}'".format(self.name, parameter)

  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
    * pos (int): result offset

    Returns:

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

    """
    pass
```

Example Search Engine Wrapper

For example, a **Picassa** (an online image sharing website) wrapper for retrieving image results is included below.

The search method must be passed a Query object (*Query*) and return a Response object (*Response*). In this example,

the processing of the results is handled by the Response class itself - as the data format from Picassa is an Atom Feed, which can be parsed automatically by the framework.

```
import urllib2

from puppy.search import SearchEngine
from puppy.model import Query, Response

class Picassa(SearchEngine):
  """
  Picassa search engine.

  Parameters:

  * resultsPerPage (int): select how many results per page
  """

  def __init__(self, service, resultsPerPage=8, **args):
    super(Picassa, self).__init__(service, **args)
    self.resultsPerPage = resultsPerPage

  def _origin(self):
    """ This overrides SearchEngine's default origin as Picassa is 1-indexed """
    return 1

  def search(self, query, offset):
    """
    Search function for Picassa.

    Parameters:

    * query (puppy.model.OpenSearch.Query)

    Returns:

    * puppy.model.OpenSearch.Response

    Raises:

    * urllib2.URLError

    """

    pos = self._origin() + offset
    userQuery = urllib2.quote(query.search_terms)
    url = "https://picasaweb.google.com/data/feed/api/all?q={0}&kind=photo".format(userQuery)

    # Add in the resultsPerPage parameter
    url += "&max-results={0}".format(self.resultsPerPage)

    # Add in pagination
    url += "&start-index={0}".format(pos)

    try:
      data = urllib2.urlopen(url)
      return Response.parse_feed(data.read())
    except urllib2.URLError, e:
      print "Error in Search Service: Picassa search failed"
```

Note, in the above example, what needs to be done to conform to the SearchEngine standard and how to construct a URL to get results from the external service.

4.3.2 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 the position for any offset/pagination parameters in the request to an external service (in the Picassa example the url variable is this request):

```
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
```

4.3.3 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 then the wrapper itself will need to parse them. The result of this parsing must be a PuppyIR response object (for more see: *Response*), with all the standard fields required by the OpenSearch standard.

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

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

CHAPTER

FIVE

APPENDICES

5.1 Frequently Asked Questions

The following are questions regarding the usage of the PuppyIR framework; it will be added to and amended over time.

5.1.1 What's the difference between Bing/BingV2 and which should I use?

Bing uses the old version of the Bing Search API, which does not require an API key and allows applications to search all the different source types apart from videos. **BingV2** however, uses the latest version of the API, which allows for not only video searching, but also retrieval of multiple source types in one call (i.e. get Video and News results for query X). However, to use the **BingV2** wrapper you require an API key, which you get by registering as a Bing developer and adding details about your application. **BingV2** also allows for new filters to be used like only retrieving square images (amongst other options), see API reference page for the full details: *BingV2 (API Version 2.2)*. It is, therefore, dependent upon the needs our your application to dictate which wrapper you should use.

5.1.2 Why can't you limit the results from RelatedSearch in Bing and BingV2

Microsoft have not yet enabled the ability to limit the number of results retrieved for this source type (as in their other source-types). It is, therefore, down to the developer to select a subset of these results if they wish to limit the results.

5.1.3 Why can't you you search for Videos in the Bing wrapper?

The **Bing** wrapper uses the old version of Microsoft's Bing Search API; using the RSS/Atom feed output format. However, this version of the API does not support searching with the video source-type.

5.1.4 How does the offset affect which page of results is retrieved and why not, just use page number?

An offset is used in all the search engine wrappers instead of a page number, because the number of the first page of results varies between search engines: some are 0-indexed and some 1-indexed. Therefore, in order to implement a generalised method the aforementioned offset is used with *0* retrieving the first page and *1* the second (i.e. go 1 page past the first page) etc.

5.1.5 What's the difference between YouTube/YouTubeV2 and which should I use?

YouTube uses the old version of the YouTube API which allows for no customisation options. Whereas, **YouTubeV2**, uses the latest version of the YouTube API which allows for greater customisation via a variety of options like *resultsPerPage* and location based searching - see the API documentation for full details: *YouTubeV2 (API Version 2.0)*. So, if you just want a basic search service, use the **YouTube** wrapper, but, if you want to customise your search service and take advantage of the new API features, use **YouTubeV2**.

5.2 The structure and the PuppyIR repository

The PuppyIR repository is organised 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 their contents.

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 (e.g. *site_media*) are not shown, in order to make the diagrams easier to read.

5.2.1 Trunk

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 trunk's contents are summarised.

5.2.2 Trunk/Framework

This folder contains the latest version of the framework, the test suite and the documentation (both the source and compiled versions).

The framework is organised as follows:

- **build** and **setup.py**: is the build directory (for when installing the framework) and the Python script to install the framework itself.
- **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 compile 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.
  - **pipeline**: contains the search engine manager and pipeline service classes, used when developing an application using the pipeline service paradigm (see: *Paradigm 2 - One Pipeline, Many Search Engines*).

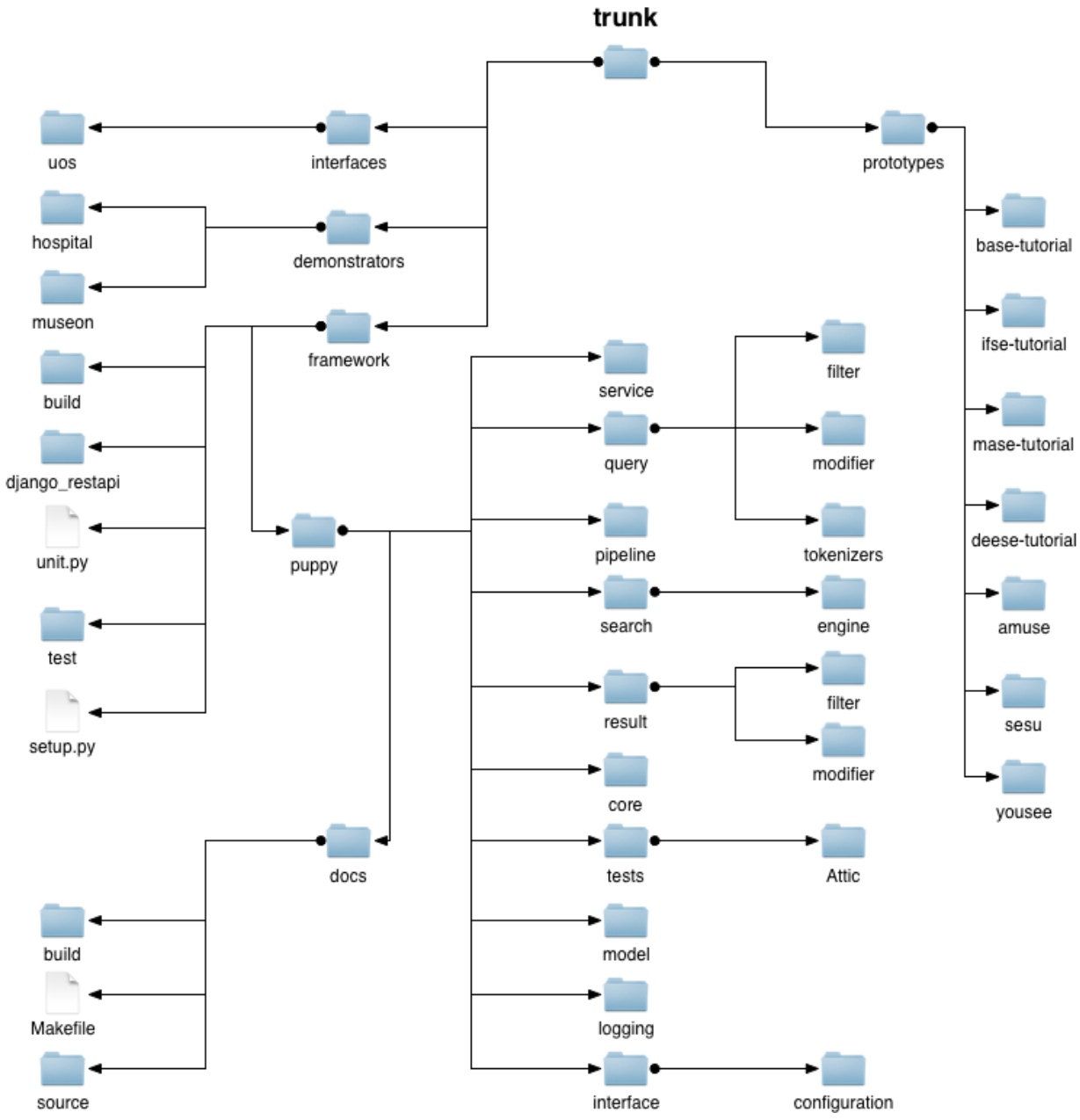

Figure 5.1: *Diagram showing the structure of the 'trunk' folder in the repository.*

  - **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, used when developing an application using the search service paradigm (see: *Paradigm 1 - One Pipeline, One Search Engine*). 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 legacy version of the test suite; the new 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, please see: *The PuppyIR Framework Test Suite* for details of this component.

5.2.3 Trunk/Demonstrators

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

Hospital Demonstrator

This demonstrator, also known as the Emma Search service (EmSe), is being built for Emma Kinderziekenhuis (EKZ), which 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 user study carried out by hospital staff, from the information centre, has uncovered that children are reluctant to engage with the physical information centre (depending instead upon a family member or carer) and so, EmSe is designed to make use of these bedside terminals to allow them to access this resource via the web.

The goals of this demonstrator are, in summary, to:

1. improve knowledge of existence and contents of the extensive library of resources, available at the information centre;
2. improve the accessibility of the information centre and its content for children;
3. expand the information content (from the Information Centre), with reference to more extensive information on the internet that is both appropriate and suitable for children.

EmSe assists the children by providing appropriate query suggestions, simplifying difficult content and filtering unsuitable content based on age appropriateness.

Musueon Demonstrator

The Museum Demonstrator creates an interactive museum visit, using advanced technologies such as multitouch tables and marker tracking, creating the basis for additional data retrieval & filtering using the PuppyIR framework. Up to four users can use a multitouch 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 (for reading and identifying the players) are installed, that when triggered, present the questions and provide feedback to answers.

Figure 5.2: *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.*

After all questions have been answered, the multitouch table provides the children with further information about the exhibits they visited.

You can view a video of this demonstrator in action by visiting: http://www.youtube.com/watch?v=b5zycfgqlKo

Prototypes

This folder contains various 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 detailed in: *Running Prototypes* - please consult this page for more.

Interfaces

This folder contains the University of Strathclyde's experimental environment on collaborative search interfaces.

5.2.4 Branches

This folder contains standalone components, related projects (made by students using the PuppyIR Framework) and unfinished/work-in-progress prototypes.

Branches 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, simplified, local copy of the PuppyIR framework.

Figure 5.3: *The Musueon demonstrator being used on multitouch tables; showing various topics.*

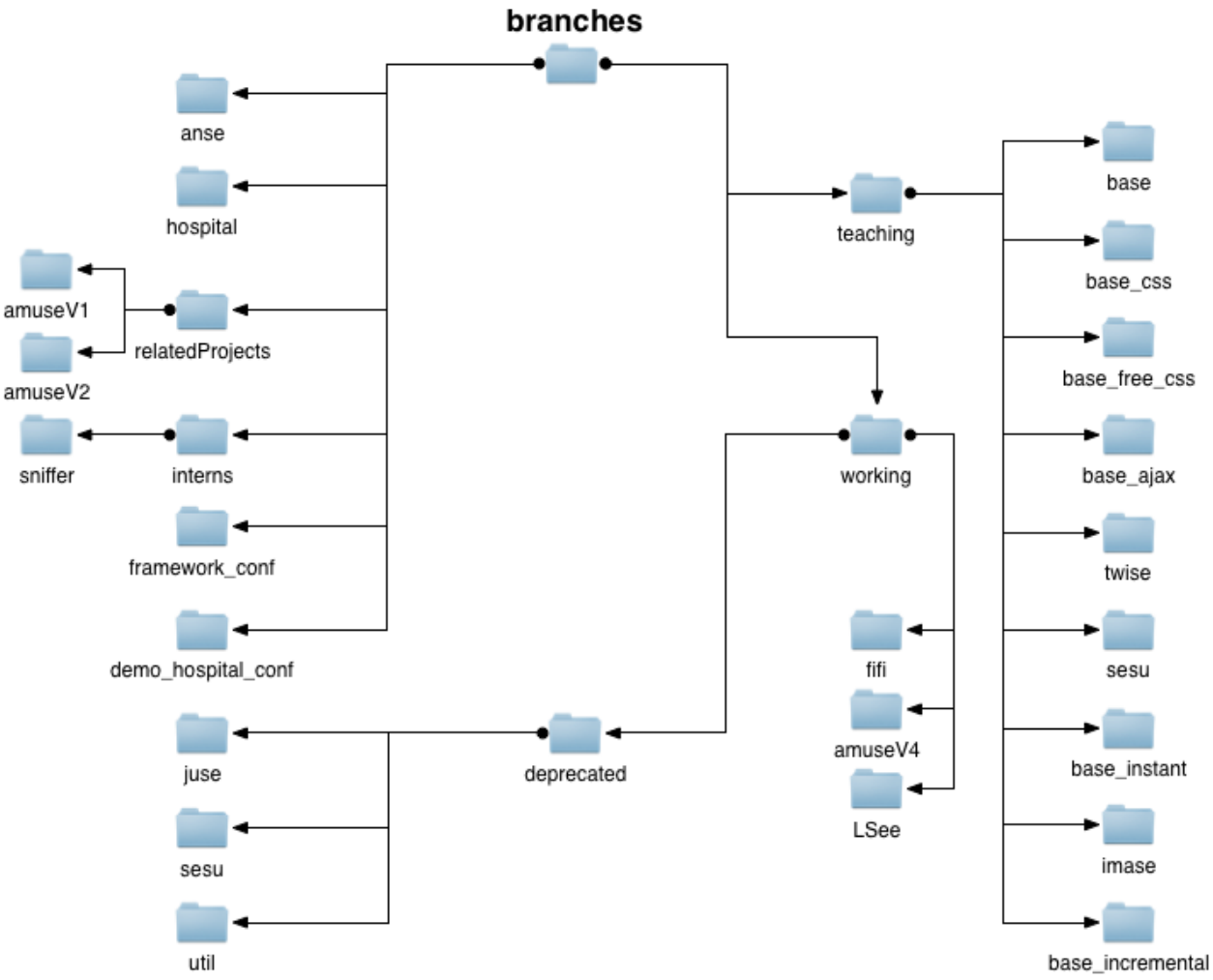

Figure 5.4: *Diagram showing the structure of the 'branches' folder in the repository.*

- **conf demos (framework and hospital)** these are early versions of a method to allow for easy configuration of these resources.
- **Interns**: a application called **sniffer**, created by a student intern 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).
- **Related Projects** this contains applications created by students using the PuppyIR framework - this folder contains its own documentation which details its contents and various other aspects, like how to install them.
- **Teaching**: this folder contains various applications created (using the PuppyIR framework) as part of the *Internet Technology* and *Distributed Information Management* courses at the University of Glasgow to teach students about web development. The individual applications it includes are:
  - **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.
  - **BaSe Incremental**: an alternate Base Ajax tutorial, for creating an Ajax based search applications using the PuppyIR Framework.
  - **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. These prototypes are described at the end of the *branches* section.
  - **Deprecated**: these prototypes use an outdated local version of the framework, **util**. SeSu does not work anymore, but JuSe does still function. Both applications and **util** are no longer supported (however, SeSu has been remade, with the latest version of the PuppyIR Framework, and can be found in *trunk*).

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 still 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 in that area. LSee (Location Search) is, functionality wise, fairly well developed but the layout and styling is very basic.

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

5.2.5 Tags

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 a older version of these components.

5.3 Known issues with the PuppyIR framework

This section details known issues with the PuppyIR framework and will be kept up-to-date, as new versions of the framework are released.

5.3.1 Python 2.6 issues

In the *Overview and background of the PuppyIR Framework* section it was noted that the framework is intended to be used with Python 2.7. The use of Python 2.6 has not been tested thoroughly and, as such, the list below of issues with it is not comprehensive (it is recommenced to switch to Python 2.7 to avoid these and other potential problems):

- The aMuSe prototype does not work, due to a change to the **fractions** library between Python 2.6 and Python 2.7 regarding the valid format(s) of data it can handle.
- Both the **YouTube** and **YouTubeV2** wrappers function, but lose all details of thumbnails in Python 2.6 due to a change in how Atom/Xml feeds are parsed.

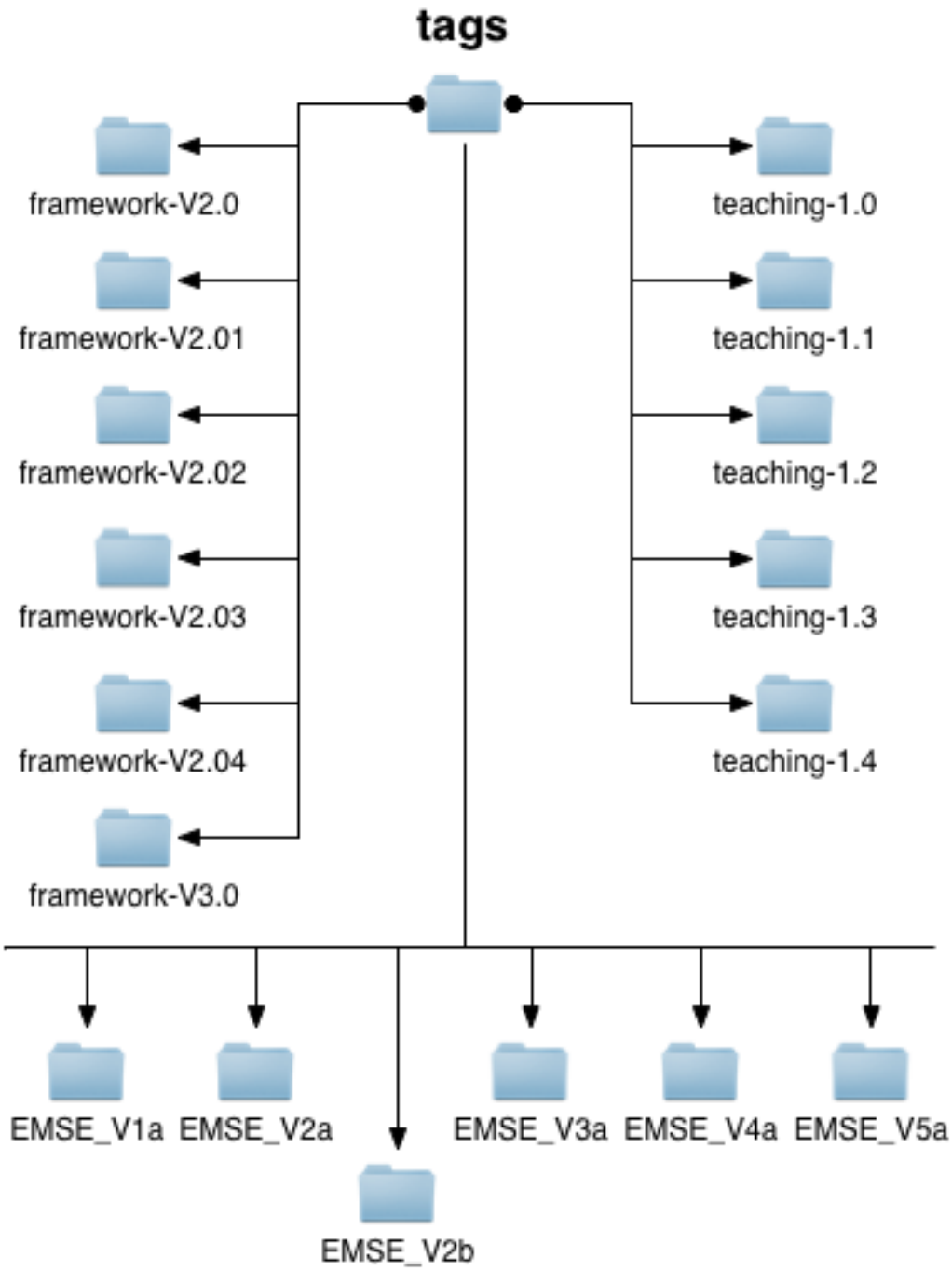

Figure 5.5: *Diagram showing the structure of the 'tags' folder in the repository.*

CHAPTER

SIX

API REFERENCE

6.1 PuppyIR API Reference

6.1.1 puppy.service

This module contains classes for building a service.

ServiceManager

class puppy.service.**ServiceManager**(*config*)
Manages a collection of search services for a PuppyIR Service

add_search_service(*obj*)
Add a search service

remove_search_service(*service_or_name*)
Removes an exisiting search service

SearchService

class puppy.service.**SearchService**(*service_manager*, *name*)
Models the configuration of a QueryFilter pipeline, Search Engine, and a ResultFilter pipeline.

add_filters(**filters*)
Add one or more filters. Detects filter type (e.g., QueryFilter, ResultModifier) and places in appropriate pipeline.

add_query_filter(*query_filter*)
Add filter to query filter pipeline.

add_query_modifier(*query_modifier*)
Add modifier to query modifier pipeline.

add_result_filter(*result_filter*)
Add filter to result filter pipeline.

add_result_modifier(*result_modifier*)
Add filter to result filter pipeline.

clear_filters()
Remove all existing filters.

replace_filters(**filters*)
Replace existing filters with new filters.

search(*query*, *offset=0*, *highlight=False*)
Search with query and result filter pipelines active.

Parameters:

•query (puppy.model.Query): search query

•offset (int): result offset

Returns:

•results (puppy.model.Response): search results

simplesearch(*query*, *offset=0*)
Search without query and result filter pipelines.

Parameters:

•query (puppy.model.Query): search query

•offset (int): result offset

Returns:

•results (puppy.model.Response): search results

6.1.2 puppy.pipeline

This module contains an alternate paradigm for creating a PuppyIR based service, where you construct one Pipeline, store multiple search engines and then either: search all or, search a specific one using the Pipeline. See: *Paradigm 2 - One Pipeline, Many Search Engines* for an explanation of this paradigm and *Pipeline Tutorial: DeeSe (Detective Search)* for details about how to go about using it to create an application.

PipelineService

class puppy.pipeline.**PipelineService**(*config*, *name*)
Models the configuration of a Pipeline (QueryFilters/Modifiers and ResultFilters/Modifiers) and the search engines using the Pipeline

add_filters(**filters*)
Add one or more filters. Detects filter type (e.g., QueryFilter, ResultModifier) and places in appropriate pipeline.

add_query_filter(*query_filter*)
Add filter to query filter pipeline.

add_query_modifier(*query_modifier*)
Add modifier to query modifier pipeline.

add_result_filter(*result_filter*)
Add filter to result filter pipeline.

add_result_modifier(*result_modifier*)
Add filter to result filter pipeline.

clear_filters()
Remove all existing filters.

replace_filters(**filters*)
Replace existing filters with new filters.

searchAll(*query*, *offset=0*)
Search all the search engines currently stored by our 'SearchEngineManager' using the query and result pipeline as currently defined.

Parameters:

•query (puppy.model.Query): search query

•offset (int): result offset

Returns:

•results_dict (dictionary of puppy.model.Response): the key being the name of the search egine and the value the reponse object

searchSpecificEngine(*query*, *searchEngineName*, *offset=0*)
Search a specific search engine only, if it's currently stored by our 'SearchEngineManager' using the query and result pipeline as currently defined.

Parameters:

•query (puppy.model.Query): search query

•searchEngineName (str): the name of the search engine to search which will be searched if it's currently stored

•offset (int): result offset

Returns:

•results_dict (dictionary of puppy.model.Response): the key being the name of the search egine and the value the reponse object

SearchEngineManager

class puppy.pipeline.**SearchEngineManager**
Manages a collection of search engines the Pipeline Manager can act upon

add_search_engine(*search_engine_name*, *search_engine*)
Adds a search engine to the Pipeline Manager if it's valid and not already stored by the Manager

clear_search_engines()
Remove all existing search engines.

get_search_engine(*search_engine_name*)
If the specified search engine is stored return it, otherwise, return None

get_search_engines()
Return all the currently stored search engines

remove_search_engine(*search_engine_name*)
Removed a search engine from the Pipeline Manager if it's currently stored by the Manager

6.1.3 puppy.search

SearchEngine

class puppy.search.**SearchEngine**(*service*, ***args*)
Abstract search engine interface.

configure_opener()
Configure urllib2 opener with network proxy if one has been added to the parent services config dictionary

search(*query*, *offset=0*)
Perform a search, retrieve the results and process them into the response format.

N.B. This should be implemented in the derived classes.

Parameters:

•query (puppy.model.Query): query object

•offset (int): result offset

Returns:

•results (puppy.model.Response): results of the search

6.1.4 puppy.search.exceptions

SearchEngineError

class puppy.search.exceptions.**SearchEngineError**(*searchEngineName*, *error*, ***extras*)
Use for exceptions in which the search engine wrapper fails - this can be for multiple reasons, for example: the lack of a proxy server in config or a search service being down. Callers should respond to this, in a way that fails gracefully.

ApiKeyError

class puppy.search.exceptions.**ApiKeyError**(*searchEngineName*, *apiFieldName*)
Use for exceptions in which the API for a wrapper, which requires one, has not been supplied. Callers should respond in such a way that the developer, it is not intended for users of an application, are aware of the issue and so can take the necessary steps to rectify the issue.

6.1.5 puppy.search.engine

Bing

class puppy.search.engine.**Bing**(*service*, *source='web'*, *adult='Strict'*, *market='en-GB'*, *resultsPerPage=10*, *lat=None*, *lon=None*, *radius=5*, *sites=None*, ***args*)
Bing search engine wrapper.

Note: you can only use location based searching with sourcetypes 'web' and 'phonebook'; however, with web, it doesn't appear to have any effect.

Parameters:

•sites: if you wish to search a specific website(s) for results

•source (str): web, image, news are the options

•adult (str): strict, i.e. safesearch not recommended to change from the default

•market (str): i.e. which area's results are prioritised more - en-gb is the UK

•resultsPerPage (int): How many results per page

•lat (double): the latitude of the place you want to search in

•lon (double): the longitude of the place you want to search in

•radius (int): the radius to retrieve results from around lat and lon; 0-250miles is the limit

BingV2 (API Version 2.2)

class puppy.search.engine.**BingV2**(*service*, *source='Web'*, *adult='Strict'*, *market='en-GB'*, *resultsPerPage=8*, *filters=None*, *sortBy=None*, *newsCategory=None*, *sites=None*, ***args*)

Bing search engine wrapper for Version 2.2 of the API - allowing for a large variety of source types to be searched.

One of the key advantages of using this wrapper is using the new features and also being able to use multiple sources to create a mash-up. i.e. source="Web+Image" gets results from the web and also image search services.

You must include your application's Bing ID in your service manage config to use this service. It should be under the identifier "bing_api_key"

If you use the 'Spell' source, then you must set the 'market' parameter to match the language you are querying in i.e. English UK set Market to en-gb or Dutch set it to nl-nl

Parameters:

•source (str): what source the results should come from, valid options are: Web, News, Video, Image, Spell, RelatedSearch.

•adult (str): Strict is the default, not recommended to change this

•market (str): For UK: en-GB, For Netherlands: nl-NL etc

•resultsPerPage (int): How many results per page

– 'Image' and 'Video' only – * filters (str): filter options split up by '+' you can only have one of each type see Bing API documentation for what these are

– 'Video' and 'News' only – * sortBy (str): sort news by either 'Date' or 'Relevance'

– 'News' only – * newsCategory (str): what sort of news is wanted - see BingAPI for list of options, for example: 'rt_ScienceAndTechnology'

Digg

class puppy.search.engine.**Digg**(*service*, *resultsPerPage=8*, *sort=None*, *topic=None*, *media='all'*, *max_date=None*, *min_date=None*, ***args*)

Digg search engine wrapper.

Parameters:

•resultsPerPage (int): How many results per page

•sort (str): how to sort results (see Digg site for a list of the options) an example is 'submit_date-desc' to sort via the item's submit date

•topic (str): restrict the search to a specific topic (see Digg site for a list of them)

•media (str): options are: 'all', 'news', 'videos', 'images'

•max_date (unix timestamp - converted to str): latest date results returned were posted

•min_date (unix timestamp - converted to str): earliest date results returned were posted

EmmaSearch

class puppy.search.engine.**EmmaSearch**(*service*, *age='v'*, *resultsPerPage=10*, ***args*)

EmmaSearch search engine.

Parameters:

•age (str): values - 'v' for adults (shows all 'a' and 'k' results too), 'a' for teenagers, and 'k' for children

•resultsPerPage (int): How many results per page - the default for the emma search service is 10

EmmaSearch SQL Server version

A new version of the above (EmmaSearch) wrapper allowing for searching the Emma Hospital database using a Microsoft's SQL server.

Due to this SQL Server import being an extra (see the installation section for details about installing it), rather than required, you cannot import this wrapper from '*puppy.search.engine*' like the above wrappers; you import them using the code below:

```
from puppy.search.engine.emmasearchMSSQL import EmmaSearchMSSQL
```

Flickr

class puppy.search.engine.**Flickr**(*service*, *sortBy='relevance'*, *safeSearch=3*, *mediaType='photos'*, *resultsPerPage=8*, *bbox=None*, ***args*)

Flickr search engine.

You must include your application's Flickr ID in your service manage config to use this service it should be under the identifier "flickr_api_key"

Parameters:

•sortBy (str): how we sort results, default is relevance see Flickr API for more details

•safeSearch (int): default is 3, i.e. strict, not recommended to change this

•mediaType (str): all, photos, videos are the options

•resultsPerPage (int): How many results per page

•bbox (str): replace the names with the values of the corners of the bounding box 'swLongitude,swLatitude,neLongitude,neLatitude'

Google Geocode

class puppy.search.engine.**GoogleGeocode**(*service*, *sensor='false'*, ***args*)

GoogleGeocode search service.

Parameters:

•sensor(str): does your device have a GPS sensor or not, not recommended to change from 'false' but the other option is, naturally, 'true' - must be lowercase

Google (depreciated)

class puppy.search.engine.**Google**(*service*, ***args*)

Google search engine.

Google have regrettfully retired this search api

Code is left here for reference purposes

Google Books

class puppy.search.engine.**GoogleBooks**(*service*, *resultsPerPage=8*, *langRestrict=None*, *filter=None*, *orderBy='relevance'*, *printType=None*, ***args*)

Google's Books search engine api.

See Google's documentation for how to specify advanced queries e.g. Hobbit+inauthor:Tolkien

Parameters:

•resultsPerPage (int): How many results per page

•langRestrict (str): restrict results to a certain language i.e. 'en' for English

•filter (str): filter volumes by type/availabilty, valid values - 'partial', 'full', 'free-ebooks', 'paid-ebooks', 'ebooks'

•orderBy (str): order either by 'relevance' or 'newest'

•printType (str): 'all', 'books' or 'magazines' restrict the results to either all or one of the preceding types of media only

Guardian

class puppy.search.engine.**Guardian**(*service*, *orderBy='newest'*, ***args*)

Guardian search engine api for searching for news stories.

Warning: 'StandFirst' is the result field used for description; it is a form of abstract for the news story. It can however, contain html tags and so when processing these results in an application this needs to be taken into account.

Parameters:

•orderBy (str): the options are - 'newest', 'oldest' and 'relevance'

iTunes

class puppy.search.engine.**ITunes**(*service*, *country='gb'*, *lang='en_gb'*, *media=None*, *resultsPerPage=8*, *explicit=False*, ***args*)

iTunes search engine wrapper - allowing for Track, Album and Artist search results to be retrieved

If you change either lang or country change the other variable to match i.e. change lang to 'en_gb' you should also change country to 'gb' to match or vice-versa.

Parameters:

•country (str): Which iTunes store to search i.e. 'gb' for the UK and 'us' for the USA etc

•lang (str): the language the results should be returned in

•media(str): the media type you want to search for (see iTunes documentation for others e.g. 'movie' etc)

•resultsPerPage (int): How many results per page

•explicit (boolean): Do we want to return results marked as including explicit content (not recommended to change this)

LastFM

class puppy.search.engine.**LastFm**(*service*, *source='track'*, *resultsPerPage=8*, *artist=None*, ***args*)
LastFM search engine wrapper - allowing for Track, Album and Artist search results to be retrieved

You must include your application's LastFM ID in your service manage config to use this service. It should be under the identifier "last_fm_api_key"

Parameters:

•source (str): What to search for, valid types: 'track', 'album' and 'artist'

•resultsPerPage (int): How many results per page

– 'Track' Only Parameters – * artist (str): the artist for the tracks you are searching for

OpenSearch

class puppy.search.engine.**OpenSearch**(*service*, *url*, ***args*)
OpenSearch search engine.

Picassa

class puppy.search.engine.**Picassa**(*service*, *resultsPerPage=8*, *access='public'*, *kind='photo'*, ***args*)
Picassa search engine.

Parameters:

•resultsPerPage (int): select how many results per page

•access (str): public, private (it is not recommended to change to private), all, visible

•kind (str): photo is the only working option

Rotten Tomatoes

class puppy.search.engine.**RottenTomatoes**(*service*, *resultsPerPage=8*, ***args*)
Rotten Tomatoes search engine.

You must include your application's Rotten Tomatoes ID in your service manage config to use this service it should be under the identifier "rotten_tomatoes_api_key"

Parameters:

•resultsPerPage (int): How many results per page

SimpleWikipedia

class puppy.search.engine.**SimpleWikipedia**(*service*, *resultsPerPage=8*, ***args*)
Simple Wikipedia search engine.

Parameters:

•resultsPerPage (int): How many results per page - note with Wiki only one page of results is returned.

Solr

class puppy.search.engine.**Solr**(*service*, *url*, ***args*)
Solr search engine.

SoundCloud

class puppy.search.engine.**SoundCloud**(*service*, *resultsPerPage=8*, *order=None*, *tags=None*, *filter=None*, *genres=None*, *types=None*, *bpmFilter=None*, *durationFilter=None*, *createdFilter=None*, ***args*)
SoundCliud search engine wrapper for a music sharing application allowing the searching for tracks.

You must include your api key for Wordnik in your service manage config to use this service. It should be under the identifier "soundcloud_api_key"

Parameters:

•resultsPerPage (int): the number of results to return for a search query

•order (str): the order to return results in, valid values are 'created_at' and 'hotness' (this later one being popularity of tracks)

•tags (str): a comma separated string of tags to look for along with the query

•filter (str): filter via the access category, valid values are: 'all', 'public', 'private', 'streamable', 'downloadable'

•genres (str): a comma separated string of genres to look for along with the query (see the SoundCloud site for a list of genres)

•types (str): a comma separated string of types of track to look for along with the query (see the SoundCloud site for a list of types - examples are 'live' or 'demo')

•bpmFilter (dict): filters via beats per minute, with the fields being 'from' and 'to' their values both being ints

•durationFilter (dict): filters via duration of the track, with the fields being 'from' and 'to' their values both being ints with the units being milliseconds

•createdFilter (dict): filters via when the track was created, with the fields being a string of format: 'yyyy-mm-dd hh:mm:ss'

Spotify

class puppy.search.engine.**Spotify**(*service*, *source='tracks'*, ***args*)
Spotify search engine.

Parameters:

•source (str): what sort of results should be returned, the options are: 'tracks', 'albums', 'artists'

Twitter

class puppy.search.engine.**Twitter**(*service*, *language='en'*, *type='mixed'*, *geocode=None*, *resultsPerPage=9*, *includeEntities=False*, ***args*)

Twitter search engine.

Geocode format is: latitude,longitude,radius - for example: '37.781157,-122.398720,1mi'

Parameters:

- language (str): en = English, de = German etc
- type (str): what sort of results to get can be - mixed, recent, popular
- geocode (str): to get queries around a specific location
- includeEntities (boolean): if this is true then a lot of meta-data is included (mentions, associated images, associated urls)
- resultsPerPage (int): results per page

WebSpellChecker

Register for an API key here: http://www.webservius.com/services/spellcheck/spellcheck

class puppy.search.engine.**WebSpellChecker**(*service*, *language='en_GB'*, ***args*)

Web Spell Checker's search engine api.

You must include your application's Web Spell Checker Api key in your service manager config to use this service It should be under the identifier "web_spell_api_key"

Parameters:

- language (str): the language/dictionary to check again i.e. 'en_US' for American English, 'nl_NL' for Dutch etc (this is case sensative)

Wikipedia

class puppy.search.engine.**Wikipedia**(*service*, *resultsPerPage=8*, *wikiLanguage='en'*, ***args*)

Wikipedia search engine.

Parameters:

- resultsPerPage (int): How many results per page - note with Wiki only one page of results is returned.
- wikiLanguage(str): which wiki api you want to search, default is en (English), nl (Dutch) is another example

Wordnik

class puppy.search.engine.**Wordnik**(*service*, *source='Definitions'*, *resultsPerPage=8*, *sourceDictionaries=None*, ***args*)

Worknik search engine wrapper for their dictionary based API. This wrapper allows for searching for spelling corrections, examples of the usage of a word (in web results), and also definitions for a word.

This API is only for English however, other languages are not supported.

You must include your api key for Wordnik in your service manage config to use this service. It should be under the identifier "wordnik_api_key"

With sourceDictionaries (see below) you can select multiple values i.e. ahd,webster but this will just return the first definition from ahd or if it doesn't have one from webster

Parameters:

- source (str): what source the results should come from, valid options are: 'Suggestions', 'Examples', 'Definitions'
- resultsPerPage (int): How many (the maximum number) results to return

– Definitions Only Parameters – * sourceDictionaries (str): the dictionary to search, if blank it defaults to the first definition. Other options are: 'all', 'ahd', 'century', 'wiktionary', 'webster', 'wordnet'

Yahoo

class puppy.search.engine.**Yahoo**(*service*, ***args*)

Yahoo search engine.

You must include your application's Yahoo ID in your service manage config to use this service. It should be under the identifier "yahoo_api_key"

YouTube

class puppy.search.engine.**YouTube**(*service*, ***args*)

YouTube search engine.

YouTubeV2 (API Version 2.0)

class puppy.search.engine.**YouTubeV2**(*service*, *resultsPerPage=8*, *safeSearch='strict'*, *orderBy='relevance'*, *format=None*, *location=None*, *locationRadius=None*, *onlyLocation=False*, ***args*)

YouTube search engine API version 2.

The orderBy parameter allows results to be filtered by their language relevence - see below for more.

N.B. in the text below replace <languageCode> with a code i.e. English: 'en', Dutch: 'nl' depending upon your applications needs.

Parameters:

- resultsPerPage (int): results per page
- safeSearch (str) : default is strict it's not recommended to change this
- orderBy: (str) rating, viewCount, relevance, relevance_lang_<languageCode>
- format (int): this defines if videos must conform to a standard for example 5 means only videos that can be embedded
- location (str): defines the location the videos should be from, in the format 'lat,lon'
- **locationRadius (str): format is '<radius><unit>' the radius around the location, within which results should be retur**
  the valid units are: m, km, ft and mi
- onlyLocation (boolean): only return results with a location (i.e. a geotag)

6.1.6 Whoosh wrappers

The following two wrappers both require Whoosh to be installed, for instructions for installing Whoosh see *Requirements and Installation*.

Due to Whoosh being an extra, rather than required, you cannot import them from '*puppy.search.engine*' like the above wrappers; you import them using the code below:

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

Whoosh Query Engine

class puppy.search.engine.whooshQueryEngine.**WhooshQueryEngine**(*service*, *whoosh_query_index_dir=''*, *resultsPerPage=8*, ***args*)

Whoosh Query log search engine.

Parameters:

- resultsPerPage (int): select how many results per page
- whoosh_query_index_dir (str): the absolute path for where you want queries indexed at

Whoosh Query Suggest Engine

class puppy.search.engine.whooshQuerySuggestEngine.**WhooshQuerySuggestEngine**(*service*, *whoosh_query_index_di* *re-* *sultsPer-* *Page=8*, ***args*)

Whoosh Query log search engine.

Paramters:

- resultsPerPage (int): select how many results per page
- whoosh_query_index_dir (str): the absolute path for where you want queries indexed at

6.1.7 puppy.model

Response

class puppy.model.**Response**(*results={}*)

Data model for search results. Response has four main attributes:

- feed: dictionary of information about the search results {title, * description, etc}
- entries: list of search results [{title, link, summary, etc}, ...]
- namespaces: list of namespaces ["http://a9.com/-/spec/opensearch/1.1/", * ...]
- version: source type of orginal results "rss/atom/json"

get_itemsperpage()
Returns the number of results per page, as reported by the search engine (usually, 10, except for Google, 8)

This number is used mainly by page algorithms.

Returns:

•opensearch_itemsperpage: the itemsperpage value

get_startindex()
Returns the start item for the current "page", as reported by the search engine. It is usually 0 or items per page * page number

This number is used mainly by page algorithms.

Returns:

•opensearch_startindex: the startindex value

get_totalresults()
Returns the number total of results, as reported by the search engine.

This number is used mainly by page algorithms.

Returns:

•opensearch_totalresults: the total_results value

static parse_feed(*xml_feed*)
Parses a RSS/ATOM feed of Opensearch results

static parse_json_suggestions(*json_doc*)
Parse a JSON document of Opensearch suggestions

static parse_xml_suggestions(*xml_doc*)
Parse a XML document of Opensearch suggestions

to_atom()
Creates an XML from a OpenSearch Response.

Returns:

•response_xml (str): OpenSearch Response as an ATOM feed

to_json()
Creates JSON from a Response object.

Returns:

•response_json (str): Response as JSON

to_rss()
Creates an RSS feed from a Response object.

Returns:

•response_xml (str): Response as RSS feed

Query

class puppy.model.**Query**(*search_terms*)
OpenSearch Query.

Models an OpenSearch Query element.

See: http://www.opensearch.org/Specifications/OpenSearch/1.1#OpenSearch_Query_element

static parse_xml (*oss_xml*)
Parse OpenSearch Query XML.

Parameters:

•oss_xml (str): OpenSearch Query XML

Returns:

•puppy.model.OpenSearch.Query

TODO code Query.parse_xml()

write_xml ()
Creates XML for OpenSearch Query.

Returns:

•query_xml (str): OpenSearch Query as XML

TODO code Query.write_xml()

Description

class puppy.model.**Description**
OpenSearch Description.

Models an OpenSearch Description document.

See: http://www.opensearch.org/Specifications/OpenSearch/1.1#OpenSearch_description_document

static parse_xml (*oss_xml*)
Parse OpenSearch Description XML.

Parameters:

•oss_xml (str): OpenSearch Description XML

Returns:

•puppy.model.OpenSearch.Description

TODO code Description.parse_xml()

write_xml ()
Creates XML for an OpenSearch Description document.

Returns:

•description_xml (str): OpenSearch Description document as XML

TODO code Description.write_xml()

6.1.8 puppy.query

QueryFilter

class puppy.query.**QueryFilter** (*order=0*)
Base class for filters that can reject queries, e.g., by detecting profanity.

QueryModifier

class puppy.query.**QueryModifier**(*order=0*)
Base class for all query modifiers

6.1.9 puppy.query.exceptions

QueryRejectionError

class puppy.query.exceptions.**QueryRejectionError**
Raise when a filter rejects a query, e.g., because profanity is detected.

QueryFilterError

class puppy.query.exceptions.**QueryFilterError**
Use for exceptions in which the filter operationally failed and the filter's function cannot be realized. Callers should respond to this as if a modification or rejection decision cannot be made, as opposed to puppy.query.QueryRejectionError, in which case the query should not be issued.

QueryModifierError

class puppy.query.exceptions.**QueryModifierError**
Use for exceptions in which the modifier operationally failed and the modifier's function cannot be realized. Callers should respond to this as if a modification or rejection decision cannot be made, as opposed to puppy.query.QueryRejectionError, in which case the query should not be issued.

6.1.10 puppy.query.filter

BlackListFilter

class puppy.query.filter.**BlackListFilter**(*order=0*, *terms=''*)
The BlackList filter looks at the query to check if any terms are contained within the black list if so, they are rejected.

Parameters:

- order (int): filter precedence
- terms: a string containing all the blacklisted terms separated by spaces i.e. ' '

WDYL Profanity Filter

class puppy.query.filter.**WdylProfanityQueryFilter**(*order=0*)
Rejects queries containing profanity using WDYL (by Google).

What this does is query the service, which returns a JSON response of true or false depending upon the presence, or not, of profanity.

Warning: there is a marked delay in waiting for a response from this service - overuse can lead to poor performance.

Parameters:

•order (int): filter precedence

SuggestionFilter

class puppy.query.filter.**SuggestionFilter**(*order=0*)
Creates a set of suggestions based upon the query search terms.

As of July 2011, Sergio's web service no longer responds and is therefore not usable.

Paramters:

•order (int): filter precedence

WhooshQueryLogger

About the Whoosh Query Logger

The Whoosh Query Logger, like the search engine wrappers for Whoosh, requires Whoosh to be installed, for instructions for installing Whoosh see *Requirements and Installation*.

Due to Whoosh being an extra, rather than required, you cannot import it from '*puppy.query.filter*' like the above filters; you import the Whoosh Query Logger using the code below:

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

class puppy.query.filter.whooshQueryLogger.**WhooshQueryLogger**(*order=0*, *whoosh_query_index_dir=''*, *unique=True*)
Logs the queries in a Whoosh Index, Creates a Whoosh Index to store queries if there is no index in the dir given with a Schema(title=ID(unique=True, stored=True), content=TEXT(stored=True), ncontent=NGRAM(stored=True), issued=DATETIME(stored=True)) Parameters:

•order (int): filter precedence

•whoosh_query_index_dir (string): path to the directory of the index

•unique (boolean): indicates whether all queries are stored, or only unique queries (i.e. if unique=True)

6.1.11 puppy.query.modifier

SpellingModifier

class puppy.query.modifier.**SpellingCorrectingModifier**(*order=0*)
This modifies queries by replacing mispelt words with the first "correct" spelling found.

Parameters:

•order (int): modifier precedence

•language (string): this defines which dictionary to use, it defaults to en_US - change this as required

Warning: this requires the PyEnchant library to be installed

TermExpansionModifier

class puppy.query.modifier.**TermExpansionModifier**(*order=0*, *terms=''*)
Expands original query terms with extra terms.

Parameters:

•order (int): modifier precedence

•terms (string): the terms to be appended to the query

KidsModifier

class puppy.query.modifier.**KidsModifier**(*order=0*, *modifiers=None*)
Base class for QueryModifiers aiming to modify queries to be more child-directed, e.g., appending for kids to query, creating Q -> Q. After modification, the Google Suggest service is checked for the presence of Q; if it exists as a frequenty query, Q is returned to the caller; otherwise, Q (the original query) is returned (hence a null operation).

KidifyQueryModifier

class puppy.query.modifier.**KidsModifier**(*order=0*, *modifiers=None*)
Base class for QueryModifiers aiming to modify queries to be more child-directed, e.g., appending for kids to query, creating Q -> Q. After modification, the Google Suggest service is checked for the presence of Q; if it exists as a frequenty query, Q is returned to the caller; otherwise, Q (the original query) is returned (hence a null operation).

6.1.12 puppy.result

ResultFilter

class puppy.result.**ResultFilter**(*order=0*)
Abstract result filter.

ResultModifier

class puppy.result.**ResultModifier**(*order=0*)
Change result.

6.1.13 puppy.result.exceptions

ResultFilterError

class puppy.result.exceptions.**ResultFilterError**
Use for exceptions in which the filter operationally failed and the filter's function cannot be realized. Callers should respond to this as if a rejection decision cannot be made.

ResultModifierError

class puppy.result.exceptions.**ResultModifierError**
Use for exceptions in which the modifier operationally failed and the modifier's function cannot be realized. Callers should respond to this as if a modification cannot be made to the result.

6.1.14 puppy.result.filter

Age Filter

class puppy.result.filter.**AgeFilter**(*age*, *ageField=None*, *ageTolerance=3*, *minAgeField='minAge'*, *maxAgeField='maxAge'*, *order=0*, *rejectUnclassified=False*)
Filters search results based on either a specific age or if the age is within an age range defined by the result.

Note: there is no default value for 'age' it must be passed to this filter so that it can be customised for the application using it.

Options:

- order (int): filter precedence
- age (integer) : the age of the user the results should be filtered for
- ageField (str) : the field name for the age in the results
- ageTolerance (int): if results just have an age field this defines the tolerance for accepting results i.e. within 3 years of the 'age' parameter - must be >= 0
- maxAgeField (str) : the field name for the maximum age in the results
- minAgeField (str) : the field name for the minimum age (if used)
- rejectUnclassified (boolean): if set to true results without an age classificiation will be rejected automatically

Duplicate Filter

class puppy.result.filter.**DuplicateFilter**(*order=0*, *existingResults=*[ ])
Filters search results and rejects ones already stored by an application. This is done by default by checking the link field of new results against a list of ones currently stored by the application. If found, they are rejected.

Options:

- order (int): defines when, in the pipeline, this filter will be executed
- existing results (list of str): urls already stored in the application - we want to avoid getting these again.

ExclusionFilter

class puppy.result.filter.**ExclusionFilter**(*order=0*, *terms=''*, *customFields=*[ ])
Filters search results based on a list of words to exclude, if any of these are found the result in question is rejected.

Options:

- order (int): defines when, in the pipeline, this filter will be executed
- terms (str): terms that, if appearing in the result, will cause it to be rejected - separated by "+'s"

•customFields (list of str): extra fields in the results to filter with the exclusion list - depedendent upon their existence in the search service results

ProfanityFilter

class puppy.result.filter.**WdylProfanityFilter**(*order=0*, *customFields=*[])
Filters results with profanity in them by using the wsdl service.

Pros:

- no hardcoded blacklist. they do the effort in keeping the service effective

Cons:

- URL call. This can mean delay. Effort should be made to parallelize the pipeline so that this effect is minimal.

Parameters:

•order (int): filter precedence

•customFields (list of str): extra fields in the results to filter with the exclusion list - depedendent upon their existence in the search service results

SuitabilityFilter

This filter evaluates a result on its suitability for children by assigning it a score of 0 (unsuitable) to 1.0 (100% suitable). For an example of how to use this filter check out the SeSu prototype - see *Running Prototypes* for details on how to install and run this prototype.

N.B. this filter requires Java to be installed and present on the system path (see: *Requirements and Installation* for more).

class puppy.result.filter.**SuitabilityFilter**(*order=0*, *threshold=0.0*)
Filters search results based on the results' suitability for children.

Parameters:

•order (int): filter precedence

•threshold (double): confidence score to accept a page (e.g. 0.5)

6.1.15 puppy.result.modifier

BlackListModifier

class puppy.result.modifier.**BlackListResultModifier**(*order=0*, *terms=''*, *customFields=*[])
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 ***

URLDecorator

class puppy.result.modifier.**URLDecorator**(*args*, *order=0*)
Decorates links to search results with given pre- and suffixes, returning [prefix]+url+[suffix].

6.1.16 puppy.logging

QueryLogger

class puppy.logging.**QueryLogger**(*search_service*, *log_mode=0*, *log_dir=None*, *log_period='midnight'*, *log_maxbytes=1000000000*)
Logs queries for a SearchService.

The QueryLogger will log all queries submitted to a SearchService, sending them to:

1.current directory, if there is no given log_dir

2.specific directory, if a log_dir filepath is given (by constructor or config)

The QueryLogger has five logging modes:

1.OneBigFile - single file that grows endlessly

2.Rotational - files rotate when log file size is = 1GB

3.Timed - files rotate every day at midnight

4.Permanent Rotating - files rate when the log file size is reached taking a unique name for each new log

5.Gzip Permanent Rotating - same as above by using Gz compression

create_logger()
Create a new logger with a specific handler

get_log_dir()
Find the log_dir if none was passed in the constructor.

Checks the service config files, then defaults to creating a log directory in the current working directory

log(*query*, *processed=False*)
logs a query using a simple [ISO Timestamp, Query Terms] format

EventLogger

class puppy.logging.**EventLogger**(*application_name*, *log_mode=0*, *log_dir=None*, *log_period='midnight'*, *log_maxbytes=1000000000*)
The EventLogger will log all events submitted to it from an application (either standalone or Django)

1.current directory, if there is no given log_dir

2.specific directory, if a log_dir filepath is given by the constructor

The EventLogger has three logging modes:

1.OneBigFile - single file that grows endlessly

2.Rotational - files rotate when log file size is = 1GB by default; can be changed via log_maxbytes

3.Timed - files rotate every day at midnight

4.Permanent Rotating - files rate when the log file size is reached taking a unique name for each new log

5.Gzip Permanent Rotating - same as above by using Gz compression

create_logger()
Create a new logger with a specific handler

get_log_dir(*log_dir*)
Works out what the log directory will be. There are three cases:

1.A log dir is given by the constructor and exits - use it

2.A log dir is given by does not exist - make it and use it

3.A log dir is not given then create one from current path

log(*identifier*, *action*, ***data*)
Logs a query using a simple [ISO Timestamp, Identifier, Action, Data] format

- Identifier (str): what identifies this log entry to a user i.e. IP address, Cookie Number etc
- Action (str): the action the user has done i.e. page request
- Data (str): associated data to the action done

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

INDEX

Y