1 Introduction

Heimdall began its existence as a Google Summer of Code project in 2012 for xbmc media center. The goal of the project was to research the possibility to create a new engine which would handle the information extraction more generically (in terms of media content) and more efficiently. Another goal was to create something which was not bound to xbmc, so it could be used by the entire ecosystem.

2 Goal

Generalized adding a new media type should be trivial, no part of the core should be bound by media types.

Dynamic fields what fields is of interest is not tied to the engine, any scraper may add metadata as it sees fit. The user of the engine (xbmc, skinners etc) may choose what data it understands but scrapers can emit all type of data it wants.

Parallelism as much as possible needs to be parallelism friendly, ideally not only between files but all parts of scraping of a file too.

3 Nomenclature

Subject The entity targeted to find more information about.

Object Can be either a literal or a url locating a subject. So it can be loosely linked data only associated with one subject or it can be another subject.

Predicate A link/edge between a subject and object

Subject, predicate and object are the same as those of the W3C rdf [2] and semantic web [1] [3].

Metadata The term is in short the same as subject, predicate and object. Usually when using the term metadata however its only speaking of a single subject and all of its predicate and objects, or a single subject with a single predicate, object.

Resource Anything holding data, which could be used to extract metadata from. A resource must be identified via URL and is a resource in the same definitions as that of the W3C standard in regards to URI and URL [6] [5].

4 Design

Heimdall uses constraints, both soft and hard, to create a pipeline for a given subject. The subject given to Heimdall may expose differing properties and the engine will create a suitable pipeline during the extraction process. To find the constraints Heimdall will use the subjects predicate and objects along with what tasks supplies and demands.

As an example a task might have a *demand* on *title* of a subject, without this title the task cannot run, thus creating a hard *constraint* between this subject task and the subject. If there is no other task which can *supply* a title, the task can never run and is thus pruned from the queue. As such Heimdall will use what a task will supply to prune the subject task queue.

In Heimdall the entire extraction process is split into tasks which will run when there demands are met. These demands create a hard constraint on the subject if they are to run.

Lets assume Heimdall possesses five tasks

Fixup which will take anything which is a resource, i.e. has a URL/URI, and append other data like mimetype

Media.StreamDetails which will if resource is media, audio or video, extract information about the container e.g. duration of streams and codec of streams

Video.Guess which will based on the media url and stream details guess a more specific content type, e.g. movie or tv show episode.

TheMovieDB.FindMovie if media is movie search and link to a themoviedb resource.

TheTVDB.FindEpisode if the media is an episode search and link to a thetvdb resource.

and a client gives Heimdall a movie url

```
from heimdall.predicates import dc

foobar = {
    dc.identifier: "file:///home/SomeUser/Movie/Alien.mkv"
}
```

The Resource.Fixup task is the only one which may run since all which is given is a URL. When it has run the subject will contain a mimetype of "video/-matroska" which will allow Media.StreamDetails to run. Media.StreamDetails will for example state that that the video length is little under two hours. When duration is known Video.Guess may run and let's say it guesses correctly that its a movie the task TheTVDB.FindEpisode may be pruned and TheMovieDB.FindMovie may run. This pipeline can be viewed in figure 1. If the client already knows that the content of the file is a movie it may send this along and the pipeline could be created as shown in figure 2

Each of these tasks can be further split up in multiple stages, each stage can have any number of requirement task which Heimdall will fulfill before running each stage. A task is considered finished when all stages has run. A task may have a result which is piped to requiring tasks.

5 Coding Style

The coding style in Heimdall follows that which is set forth in PEP-8 [4]. There are a few exceptions though, namely, json styled object and list indentations are allowed. As an example:

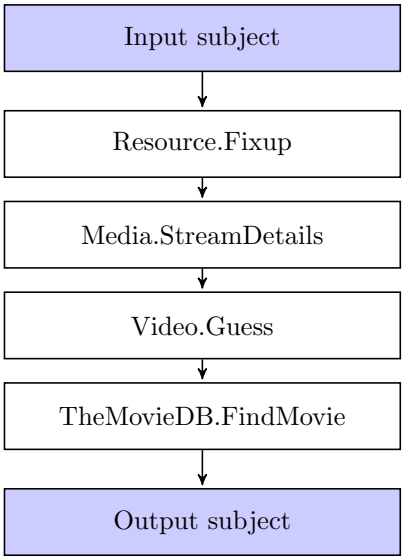

Figure 1: The sequential pipeline created by only giving URL

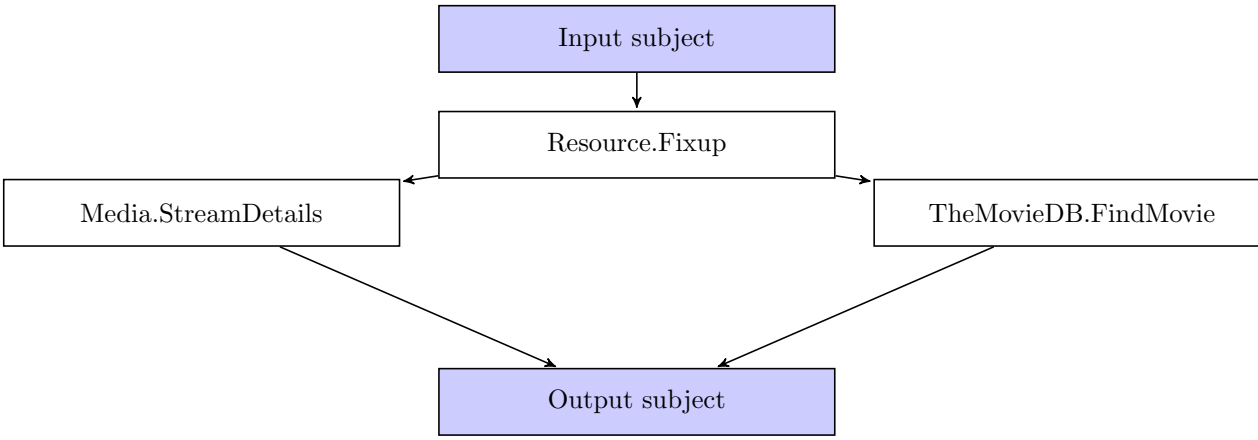

Figure 2: The parallel pipeline created by only giving URL and content

```
foobar = {
    "foo": "bar",
    "bar": "foo"
}

foobar = [
    1,
    2
]
```

References

[1] Semantic web. http://en.wikipedia.org/wiki/Semantic_Web.

[2] W3C Web Applications Working Group. Resource description framework (rdf). http://www.w3.org/RDF/.

[3] W3C Web Applications Working Group. W3c semantic web activity. http://www.w3.org/2001/sw/.

[4] Barry Warsaw Guido van Rossum. Pep 8 – style guide for python code. http://www.python.org/dev/peps/pep-0008/.

[5] W3C/IETF URI Planning Interest Group. Uris, urls, and urns: Clarifications and recommendations 1.0. http://www.w3.org/TR/uri-clarification/.

[6] URI working Group. Uniform resource locators (url). http://www.w3.org/Addressing/URL/url-spec.txt.