CLAM Documentation

version 0.5-pre-alpha

Maarten van Gompel
ILK Research Group
Tilburg center for Cognition and Communication
Tilburg University

December 12, 2010

Contents

1 Introduction **3**

2 Documentation for Service Providers **6**

2.1 Technical details 6

2.1.1 Installation 6

2.1.2 Using CLAM with Apache 2 8

2.1.3 Using CLAM with nginx 9

2.1.4 Using CLAM with other webservers 11

2.2 Architecture 11

2.3 Service configuration 11

2.3.1 Server Administration 13

2.3.2 User Authentication 13

2.3.3 Command Definition 15

2.3.4 Paradigm: Metadata, Profiles & Parameters 16

2.3.5 Parameter Specification 18

2.3.6 Profile specification 21

2.3.7 Parameter Conditions 24

2.3.8 Converters 26
2.3.9 Viewers 27
2.3.10 Working with pre-installed data 28
2.4 Wrapper script 29
2.4.1 CLAM Data API 30

3 Documentation for Service Clients **33**

A RESTful specification **35**

B Prefined Formats **40**

Chapter 1

Introduction

The Computational Linguistics Application Mediator (CLAM) allows you to quickly and transparently transform your Natural Language Processing application into a *RESTful* webservice, with which automated clients can communicate, but which at the same time also acts as a modern webapplication with which human end users can interact. CLAM takes a description of your system and wraps itself around the system, allowing clients or users to upload input files to your application, start your application with specific parameters of their choice, and download and view the output of the application. Whilst the application runs, users can monitor its status.

CLAM is set up in a universal fashion, making it flexible enough to be wrapped around a wide range of computational linguistic applications. These applications are treated as a black box, of which only the parameters, input formats, and output formats need to be described. The applications themselves need not be network-aware in any way, nor aware of CLAM, and the handling and validation of input can be taken care of by CLAM.

CLAM is entirely written in Python and is available as open source under the GNU Public License (v3). It is set up in a modular fashion and as such is easily extendable. It offers a rich API for writing clients and wrapper scripts.

The kind of applications that CLAM is intended for are Natural Language Processing applications, usually of a kind that do some processing on a text corpus. This corpus (any text file) can be uploaded by the user, or may be pre-installed for the webservice. The NLP application is usually expected to produce a certain output, which is subsequently made available through the webservice for viewing

and downloading.

The CLAM webservice is a RESTful webservice, meaning it uses the HTTP verbs GET, POST, PUT and DELETE to manipulate resources and returns responses using the HTTP response codes. The principal resource in CLAM is called a *project*. Various users can maintain various projects, each representing one specific run of the system, with particular input data, output data, and a set of configured parameters.

In addition to using HTTP Error Codes for error responses, the webservice responds in the CLAM XML format. An associated XSL stylesheet can directly transform this to xhtml in the user's browser, thus providing a standalone web application for humand end-users.

CLAM comes with an ample number of features, the most notable ones being:

- **RESTful webservice** – *CLAM is a fully RESTful webservice*
- **Webapplication** – *CLAM is also a modern "web 2.0" web application, heavily relying on technologies such as XSLT and AJAX*
- **Extensible** – *Due to a modular setup, CLAM is quite extensible*
- **Client and Data API** – *A rich Python API for writing CLAM Clients and system wrappers*
- **Authentication** – *A user-based authentication mechanism through HTTP Digest is provided*
- **Metadata and provenance data** – *There is extensive support for metadata and provenance data*
- **Automatic converters** – *Automatic converters enable conversion from an auxiliary format into the desired input format, and conversion from the produced output format into an auxiliary output format*
- **Viewers** – *Viewers enable web-based visualisation for a particular format. CLAM supports both built-in python-based viewers as well as external viewers in the form of external (non-CLAM) webservices.*

This documentation is split into two parts: a chapter for service providers, people who want to build a CLAM Webservice around their tool, and a chapter for service clients, users wanting to write automated clients to communicate with the aforemented webservice.

Development Notes
Note that at this stage, CLAM is still under development. As such, this documentation is not complete yet, but is also a work in progress.

Chapter 2

Documentation for Service Providers

2.1 Technical details

CLAM is written in Python 2.5, and is built on the webpy framework. It can run stand-alone thanks to the built-in cherrypy webserver; no additional webserver is needed to test your service. In production environments, it is however strongly recommended that CLAM is integrated into a real webserver. Supported are: Apache, nginx or lighthttpd, though others may work too.

Note that the software is designed for Unix-based systems (e.g. Linux or BSD) only.

2.1.1 Installation

The following software is required to run CLAM. These will be available in any modern Linux or BSD distribution:

- python 2.5 (or a higher 2.x version)
- python-webpy, version 0.33 or higher
- python-lxml, version 2 or higher

For development and testing, each CLAM webservice can run stand-alone on any TCP port of your choice (make sure the port is open in your firewall) using the built-in webserver. For production environments, it is strongly recommended you plug CLAM into a more advanced webserver (Apache, nginx, lighttpd).

If you want to run in the supplied test units, an additional dependency is needed: python-unittest2.

To install CLAM, simply uncompress the clam software archive in any desired target location. The following files may be of particular interest:

- `clamservice.py` – The webservice itself, the command to be invoked to start it.
- `config/` – The directory containing service configuration files. Place you service configuration here.
- `config/textstats.py` – An example configuration.
- `common/` – Common Python modules for CLAM.
- `common/parameters.py` – Parameter-type definitions.
- `common/format.py` – Format-type definitions.
- `common/data.py` – CLAM Data API.
- `common/client.py` – CLAM Client API.
- `static/style.css` – The styling for visualisation.

Starting the service in stand-alone mode is done by launching clamservice.py with the name of your service configuration. This standalone mode is intended primarily for development purposes and not recommended for production use.

```
$ ./clamservice.py clam.config.test
```

Setting up the service to be used with an already existing webserver requires some additional work. This is explained below for Apache and nginx:

2.1.2 Using CLAM with Apache 2

In order to run CLAM in Apache, you have to install and configure several files. We will be using WSGI, an interface between Apache and Python. Follow the instructions:

1. Install `mod_wsgi` for Apache 2, if not already present on the system. In Debian and Ubuntu this is available as a package named `libapache2-mod-wsgi`.

2. Next we need to write a simple WSGI-script, which is a Python script that will be invoked by the webserver. Copy `clam/config/example.wsgi` to something like `clam/config/yourservice.wsgi` and adapt the script. If CLAM is not installed in a standard location where Python can find it, make sure to explicitly specify its parent directory according to the instructions in the example.

3. Configure your service configuration file as explained in Section 2.3. Take special note of Subsection where you are instructed to configure the hostname, port, and optionally a URL prefix to use if the service is not assigned a virtualhost of its own.

4. Configure Apache to let it know about WSGI and your service. I assume the reader is acquainted with basic Apache configuration and will only elaborate on the specifics for CLAM. Adapt and add the following to any of your sites in `/etc/apache2/sites-enabled` (or optionally directly in `httpd.conf`), within any `VirtualHost` context. Here it is assumed you configured your service configuration file with `URLPREFIX` set to *"yourservice"*.

```
WSGIScriptAlias /yourservice /path/to/clam/config/yourservice.wsgi/
WSGIDaemonProcess yourservice user=proycon group=users \
    home=/path/to/clam threads=15 maximum-requests=10000
WSGIProcessGroup yourservice

Alias /yourservice/static /path/to/clam/static/
<Directory /path/to/clam/static/>
   Order deny,allow
   Allow from all
</Directory>
```

   The WSGIDaemonProcess directive goes on online, but was wrapped here for presentational purposes. Needless to say, all paths need to be adapted

according to your setup and the configuration can be extended further as desired, with for example extra authentication or more restrictive access.

5. Restart Apache

Note that we run WSGI in Daemon mode. For the specific options to the WSGIDaemonProcess directive you can check `http://code.google.com/p/modwsgi/wiki/ConfigurationDirectives#WSGIDaemonProcess`. Important settings are the user and group the daemon will run as, the home directory it will run in. The number of threads, processes, and maximum-requests can also be configured to optimise performance and system resources according to your needs.

2.1.3 Using CLAM with nginx

With nginx (version 0.8 or above), CLAM can be set up over WSGI or FastCGI. With Apache we already explored a WSGI option above, so we will now take a look at FastCGI:

1. Nginx misses a mime-type we need. Add the following line to `/etc/nginx/mime.types`:

```
text/xsl                                  xsl;
```

2. Configure your service configuration file as explained in Section 2.3. Take special note of Subsection where you are instructed to configure the hostname, port, and optionally a URL prefix to use if the service is not assigned a virtualhost of its own.

3. Make a script `start_yourservice.sh` which will start the daemon for FastCGI. Change `UID` and `GID` with user ID/group ID you intend to use. Note that the IP and port can be set to anything you like, as long as you use the same consistently throughout the configuration.

```
#!/bin/bash
spawn-fcgi -u UID -g GID -d /path/to/clam \
  -a 127.0.0.1 -p 9002 -- /path/to/clam/clamservice.py
```

4. Make a script `stop_yourservice.sh` as a convenient shortcut to stop the service again:

```
#!/bin/bash
kill `pgrep -f "python /path/to/clam/clamservice.py"`
```

5. Add and adapt the following configuration to a server in `/etc/nginx/sites-enabled`. Note that in this example we assume that `URLPREFIX` in the service configuration file is set to an empty string (or not set at all), effectively exposing CLAM at the root of the server. You may configure a `URLPREFIX` when desired. In that case, take care to update the below `location` and directives accordingly:

```
root /path/to/clam;

location / {
   fastcgi_param REQUEST_METHOD $request_method;
   fastcgi_param QUERY_STRING $query_string;
   fastcgi_param CONTENT_TYPE $content_type;
   fastcgi_param CONTENT_LENGTH $content_length;
   fastcgi_param GATEWAY_INTERFACE CGI/1.1;
   fastcgi_param SERVER_SOFTWARE nginx/$nginx_version;
   fastcgi_param REMOTE_ADDR $remote_addr;
   fastcgi_param REMOTE_PORT $remote_port;
   fastcgi_param SERVER_ADDR $server_addr;
   fastcgi_param SERVER_PORT $server_port;
   fastcgi_param SERVER_NAME $server_name;
   fastcgi_param SERVER_PROTOCOL $server_protocol;
   fastcgi_param SCRIPT_FILENAME $fastcgi_script_name;
   fastcgi_param PATH_INFO $fastcgi_script_name;
   fastcgi_pass 127.0.0.1:9002;
}

location /static/ {
   root /path/to/clam;
   if (-f $request_filename) {
      rewrite ^/static/(.*)$  /static/$1 break;
   }
}
```

6. Launch `start_yourservice.sh` and (re)start nginx.

2.1.4 Using CLAM with other webservers

You are not limited to using either Apache with WSGI or nginx with FastCGI; we tested only these two. It should also be possible to get CLAM working on other Unix based webservers, such as for example lighttpd. Although we have no CLAM-specific instructions, you may find instructions for WebPy, the framework CLAM uses, at `http://webpy.org/`, and can adapt these to CLAM.

2.2 Architecture

CLAM has a layered architecture, with at the core the NLP application(s) you want to turn into a webservice. The application itself can remain untouched and unaware of CLAM. The scheme in Figure 2.1 illustrates the various layers:

The workflow interface layer is not provided nor necessary, but shows a possible use-case.

A CLAM webservice needs the following three components from the service developer:

1. A service configuration file;
2. A wrapper script for your NLP application;
3. An NLP application.

The wrapper script is not strictly mandatory if the NLP application can be directly invoked by CLAM. However, for more complex applications, writing a wrapper script is strongly recommended, as it offers more flexibility and better integration, and allows you to keep the actual NLP application unmodified. The wrapper scripts can be seen as the "glue" between CLAM and your application, taking care of any translation steps.

2.3 Service configuration

The service configuration consists of a description of your NLP application, or rather, a description of the system wrapper script that surrounds it. It specifies

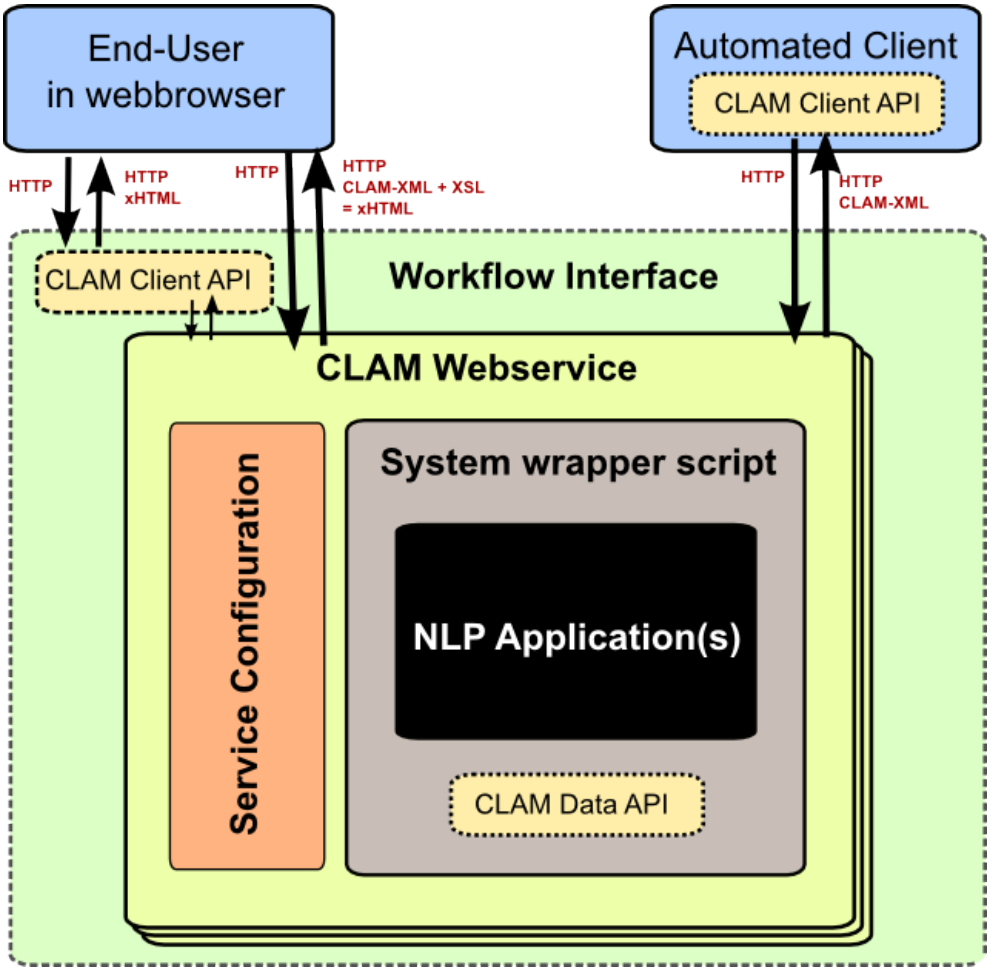

Figure 2.1: The CLAM Architecture

what parameters the system can take, and what input and output formats are expected under what circumstances. The service configuration is itself a Python script, but knowledge of Python is not essential to be able to make your own service configurations.

The server configuration files reside in the `config/` directory. Making a new webservice starts with copying the sample `defaults.py` and editing your copy. When reading this section, it may help your understanding to inspect this file alongside.

One of the first things to configure is the root path (`ROOT`). All projects will be confined to the `projects/` directory within this root path, each project having

its own subdirectory. When your NLP application or wrapper script is launched, the current working directory will be set to this project directory. Pre-installed corpora should be put in the `corpora/` directory. The `ROOT` will be automatically created upon the first run.

2.3.1 Server Administration

The hostname and port of the webserver can be configured in the service configuration file. Note that the hostname has to match exactly with what the end users will use. An attempt will be made to detect this automatically if no hostname is specified. A mismatch in the name you define and the hostname the user uses may result in unexpected behaviour[1]. CLAM comes with a built-in webserver, which will be used when invoked directly from the command-line[2].

When CLAM runs in an existing webserver without its own virtual host, it is often configured at a different URL rather than the webserver root. In this case the value of `URLPREFIX` should be configured accordingly.

2.3.2 User Authentication

Being a RESTful webservice, user authentication proceeds over HTTP itself. CLAM implements HTTP Digest Authentication, which as opposed to Basic Authentication computes a hash of the username and password client-side and transmits that hash, rather than a plaintext password.

A list of users can be defined in `USERS` in the service configuration file. This is a dictionary of usernames mapped to an md5 hash computed on the basis of the username, the system ID, and the password. Furthermore there is a setting `PROJECTS_PUBLIC`, which can be set to `True` or `False`. In the former case all projects will be open to all authenticated users; in the latter case projects will only be accessible by their owners.

User authentication is not mandatory, but for any world-accessible environment it is most strongly recommended, for obvious security reasons. Extra security may also be provided on a more global webserver level, rather than in CLAM itself.

[1] Most likely, the XSLT stylesheet will refuse to build the web application interface
[2] unless FastCGI mode is enabled

The ability to view and set parameters can be restricted to certain users. You can use the extra parameter options `allowusers=` or `denyusers=` to set this. See section 2.3.6. A common use would be to define one user to be the guest user, for instance the user named "guest", and set `denyusers=['guest']` on the parameters you do not want the guest user to use.

Development Notes
At a later stage, OpenID support will be added.

2.3.3 Command Definition

Central in the configuration file is the command that CLAM will execute. This command should start the actual NLP application, or preferably a script wrapped around it. Full shell syntax is supported. In addition there are some special variables you can use that will be automatically set by CLAM.

- `$INPUTDIRECTORY` – The absolute path to the input directory where all the input files from the user will be stored (possibly in subdirectories). This input directory is the `input/` subdirectory in the project directory.
- `$OUTPUTDIRECTORY` – The absolute path to the output directory. Your system should output all of its files here, as otherwise they are not accessible through CLAM. This output directory is the `output/` subdirectory in the project directory.
- `$STATUSFILE` – The absolute path to a status file. Your system may write a short message to this status file, indicating the current status. This message will be displayed to the user in CLAM's interface. The status file contains a full log of all status messages, thus your system should write to this file in append mode. Each status message consists of one line terminated by a newline character. The line may contain three tab delimited elements that will be automatically detected: a percentage indicating the progress until completion (two digits with a % sign), a Unix timestamp (a long number), and the status message itself (a UTF-8 string).
- `$PARAMETERS` – This variable will contain all parameter flags and the parameter values that have been selected by the user. It is recommended however to use $DATAFILE instead.
- `$DATAFILE` – The absolute path to the data file that CLAM outputs in the project directory. This data file, in CLAM XML format, contains all parameters along with their selected values. Furthermore it contains the inputformats and outputformats, and a listing of uploaded input files and/or pre-installed corpora. System wrapper scripts can read this file to obtain all necessary information, and as such this method is preferred over using $PARAMETERS. If the system wrapper script is written in Python, the

CLAM Data API can be used to read this file, requiring little effort on the part of the developer.

- `$USERNAME` – The username of the logged-in user.

Make sure the actual command is an absolute path, or that the executable is in the `$PATH` of the user `clamservice.py` will run as. Upon launch, the current working directory will be automatically set to the specific project directory. Within this directory, there will be an `input/` and `output/` directory, but use the full path as stored in `$INPUTDIRECTORY` and `$OUTPUTDIRECTORY/`. All uploaded user input will be in this input directory, and all output that users should be able to view or download, should be in this output directory. Your wrapper script and NLP tool are of course free to use any other locations on the filesystem for whatever other purposes.

2.3.4 Paradigm: Metadata, Profiles & Parameters

In order to explain how to build service configuration files for the tools you want to make into webservices, we first need to clarify the paradigm CLAM uses. We shall start with a word about metadata. Metadata is data *about* your data, i.e. data about your input and output files. Take the example of a plain text file: metadata for such a file can be for example the character encoding the text is in, and the language the text is written in. Such data is not necessarily encoded within the file itself, as is also not the case in the example of plain text files. CLAM therefore builds external metadata files for each input and output file. These files contain all metadata of the files they describe. These are stored in the CLAM Metadata XML format, a very simple and straightforward format[3] Metadata simply consists out of metadata fields and associated values.

Metadata in CLAM is tied to a particular file format (such as plain text format, CSV format, etc.). A format defines what kind of metadata it absolutely needs, but usually still offers a lot of freedom for extra metadata fields to the service provider, or even to the end user.

When a user or automated client uploads a new input file, metadata is often not available yet. The user or client is therefore asked to provide this. In the

[3]It is in essence a simple XML representation of key–value pairs. These metadata files are named `.filename.METADATA`, in which filename is the name of the file it describes and reside in the very same input/output directory.

webapplication a form is presented with all possible metadata parameters; the system will take care of generating the metadata files according to the choices made. If the service provider does not want to make use of any metadata description at all, then that is of course an option as well, though this may come at the cost of your service not providing enough information to interact with others.

In a webservice it is important to precisely define what kind of input goes in, and what kind of output goes out: this results in a deterministic and thus predictable webservice. It is also necessary to define exactly how the output metadata is based on the input metadata, if that is the case. These definitions are made in so-called *profiles*. A profile defines *input templates* and *output templates*. The input templates and output template can be seen as "slots" for certain filetypes and metadata. An analogy from childhood memory may facilitate understanding this, as shown and explained in Figure 2.2:

A profile is thus a precise specification of what output files will be produced given what input files, it specifies exactly how the metadata for the outputfiles can be constructed given the metadata of the inputfiles. The generation of metadata for output files is fully handled by CLAM, outside of your wrapper script and NLP application.

Input templates are specified in part as a collection of parameters for which the user/client is expected to choose a value in the predetermined range. Output templates are specified as a collection of "metafields", which simply assign a value, unassign a value, or copy a value from an inputtemplate or from a global parameter. Through these templates, the actual metadata can be constructed. Input templates and output templates always have a label describing their function. Upon input, this provides the means for the user to recognise and select the desired input template, and upon output, it allows the user to easily recognise the type of output file. How all this is specified exactly will be demonstrated in detail later.

In addition to input files and the associated metadata parameters, there is another source of data input: global parameters. A webservice may define a set of parameters that it takes. We will start by explaining this part in the next section.

Figure 2.2: Box and blocks analogy from childhood memory: the holes on one end correspond to input templates, the holes on the other end correspond to output templates. Imagine blocks going in through one and out through the other. The blocks themselves correspond to input or output files *with attached metadata*. Profiles describe how one or more input blocks are transformed into output blocks, which may differ in type and number.Granted, I'm stretching the analogy here; your childhood toy did not have this magic feature of course!

2.3.5 Parameter Specification

The parameters which an NLP application, or rather the wrapper script, can take, are defined in the service configuration. First of all parameters can be subdivided into parameter groups, but these serve only presentational purposes.

There are seven parameter types available, though custom types can be easily added[4]. Each parameter type is a Python class taking the following mandatory arguments:

[4]to `common/parameters.py`

1. `id` – An id for internal use only.
2. `name` – The name of this parameter, this will be shown to the user in the interface.
3. `description` – A description of this parameter, meant for the end-user.

The seven parameter types are:

- `BooleanParameter` – A parameter that can only be turned on or off, represented in the interface by a checkbox. If it is turned on, the parameter flag is included in `$PARAMETERS`, if it is turned off, it is not. If `reverse=True` is set, it will do the inverse.
- `IntegerParameter` – A parameter expecting an integer number. Use `minrange=`, and `maxrange=` to restrict the range if desired.
- `FloatParameter` – A parameter expecting a float number. Use `minrange=`, and `maxrange=` to restrict the range if desired.
- `StringParameter` – A parameter taking a string value. Use `maxlength=` if you want to restrict the maximum length.
- `TextParameter` – A parameter taking multiple lines of text.
- `ChoiceParameter` – A multiple-choice parameter. The choices must be specified as a list of `(ID, label)` tuples, in which ID is the internal value, and label the text the user sees. For example, suppose a parameter with flag `-c` is defined. `choices=[('r','red'),('g','green'),('b', 'blue)]`, and the user selects "green", then `-c g` will be added to `$PARAMETERS`. The default choice can be set with `default=`, and then the ID of the choice. If you want the user to be able to select multiple parameters, then you can set the option `multi=True`. The IDs will be concatenated together in the parameter value. A delimiter (a comma by default) can be specified with `delimiter=`. If you do not use `multi=True`, but you do want all options to be visible in one view, then you can set the option `showall=True`.
- `StaticParameter` – A parameter with a fixed immutable value. This may seem a bit of a contradiction, but it serves a purpose in forcing a parameter or metadata parameter to have a specific non-variable value.

All parameters can take the following extra keyword arguments:

- `paramflag` – The parameter flag. This flag will be added to `$PARAMETERS` when the parameter is set. Consequently, it is mandatory if you use the `$PARAMETERS` variable in your `COMMAND` definition. It is customary for parameter flags to consist of a hyphen and a letter or two hyphens and a string. Parameter flags could be for example be formed like: `-p` ,`--pages`, `--pages=`. There will be a space between the parameter flag and its value, unless it ends in a = sign or `nospace=True` is set. Multi-word string values will automatically be enclosed in quotation marks for the shell to correctly parse them. Technically, you are also allowed to specify an empty parameter flag, in which case only the value will be outputted as if it were an argument.
- `default` – Set a default value.
- `required` – Set to `True` to make this parameter required rather than optional.
- `require` – Set this to a list of parameter IDs. If this parameter is set, so must all others in this list. If not, an error will be returned.
- `forbid` – Set this to a list of parameter IDs. If this parameter is set, none of the others in the list may be set. If not, an error will be returned.
- `allowusers` – Allow only the specified lists of usernames to see and set this parameter. If unset, all users will have access. You can decide whether to use this option or `denyusers`, or to allow access for all.
- `denyusers` – Disallow the specified lists of usernames to see and set this parameter. If unset, no users are blocked from having access. You can decide whether to use this option or `allowusers`, or to allow access for all.

The following example defines a boolean parameter with a parameter flag:

```
BooleanParameter(
  id='createlexicon',
  name='Create Lexicon',
  description='Generate a separate overall lexicon?',
  paramflag='-l'
)
```

Thus, if this parameter is set, the invoked command will have `$PARAMETERS` set to `-l 1` (plus any additional parameters).

2.3.6 Profile specification

Multiple profiles may be specified, and all profiles are always assumed to be independent of each other. Dependencies should be together in one profile, as each profile describes how a certain type of input file is transformed into a certain type of output file. For each profile, you need to define input templates and output templates. All matching profiles are assumed to be delivered as promised. A profile matches if all input files according to the input templates of that profile are provided and if it generates output. If no input templates have been defined at all for a profile, then it will match as well, to allow for the option of producing output files that are not dependent on input files. A profile is allowed to mismatch, but if none of the profiles match, the system will produce an error, as it can not perform any actions.

The profile specification skeleton looks as follows. Note that there may be multiple input templates and/or multiple output templates:

```
PROFILES = [
    Profile( InputTemplate(...), OutputTemplate(...) )
]
```

The definition for `InputTemplate` takes three mandatory arguments:

1. `id` – An ID for the InputTemplate. This will be used internally and by automated clients.
2. `format` – This points to a Format class, indicating the kind of format that this inputtemplate accepts. Formats are defined in `clam/common/formats.py`. Custom formats can be added there.
3. `label` – A human readable label for the input template. This is how it will be known to users in the web application and on the basis of which they will select it.

Subsequently you may specify any of the Parameter types to indicate the accepted/required metadata. Use any of the types from Section .

After specifying any such parameters, there are some possible keyword arguments:

1. `unique` – Set to `True` or `False`, this indicates whether the input template may be used only once or multiple times. `unique=True` is the default if not specified.
2. `multi` – The logical inverse of the above; you can whichever you prefer. `multi=False` is the default if not specified.
3. `filename` – Files uploaded through this input template will receive this filename (regardless of how the original file on the client is called). If you set `multi=True` or its alias `unique=False`, insert a single # character in the filename, which will be replaced by a number in sequence. After all, we cannot have multiple files with the same name.
4. `extension` – Files uploaded through this input template are expected to have this extension, but can have whatever filename. Here it doesn't matter whether you specify the extension with or without the prefixing period. Note that in the web application, the extension is appended automatically regardless of the filename of the source file. Automated clients do have to take care to submit with the proper extension right away.

Take a look at the following example of an input template for plaintext documents for an automatic translation system:

```
InputTemplate('maininput', PlainTextFormat,"Translator input: Plain-text document",
  StaticParameter(
    id='encoding',name='Encoding',description='The character encoding of the file',
    value='utf-8'
  ),
  ChoiceParameter(
    id='language',name='Language',description='The language the text is in',
    choices=[('en','English'),('nl','Dutch'),('fr','French')]),
  ),
  extension='.txt',
  multi=True
)
```

For `OutputTemplate`, the syntax is similar. It takes the three mandatory arguments *id*, *format* and *label*, and it also takes the four keyword arguments laid out above. If no explicit filename has been specified for an output template,

then it needs to find out what name the output filename will get from another source. This other source is the input template that acts as the *parent*. The output template will thus inherit the filename from the input template that is its parent. In this way, the user may upload a particular file, and get that very same file back with the same name. If you specify `extension`, it will prepend an extra extension.

As there may be multiple input templates, it is not always clear what input template is the parent. The system will automatically select the *first* defined input template with the same value for unique/multi the output template has. If this is not what you want, you can explicitly set a parent using the `parent` keyword, which takes the value of the input template's ID.

Whereas for `InputTemplate` you can specify various parameter types, output templates work differently. Output templates define what metadata fields (metafields for short) they want to set with what values, and from where to get these values. In some situations the output file is an extension of the input file, and you want it to inherit the metadata from the input file. Set `copymetadata=True` to accomplish this: now all metadata will be inherited from the parent, but you can still make modifications.

To set (or unset) particular metadata fields you specify so-called "metafield actors". Each metafield actor sets or unsets a particular metadata attribute. There are four different types of metafield actors:

- `SetMetaField(`*`key,value`*`)` – Set metafield *key* to the specified value.
- `UnsetMetaField(`*`key [,value]`*`)` – If a value is specified: Unset this metafield if it has the specified value. If no value is specified: Unset the metafield regardless of value. This only makes sense if you set `copymetadata=True`.
- `CopyMetaField(`*`key, inputtemplate.key`*`)` – Copy metadata from one of the input template's metadata. Here *inputtemplate* is the ID of one of inputtemplates in the profile, and the *key* part is the metadata field to copy. This allows you to combine metadata from multiple input source into your output metadata.
- `ParameterMetaField(`*`key, parameter-id`*`)` – Get the value for this metadata field from a global parameter with the specified ID.

Take a look at the following example for a ficticious automatic translation system, translating to Esperanto.

```
OutputTemplate('translationoutput', PlainTextFormat,"Translator output: Plain-text document",
    CopyMetaField('encoding','maininput.encoding')
    SetMetaField('language','eo'),
    multi=True
)
```

Putting it all together, we obtain the following profile definition describing a fictitious machine translation system from English, Dutch or French to Esperanto, where the system accepts and produces UTF-8 encoded plain-text files.

```
PROFILES = [
  Profile(
    InputTemplate('maininput', PlainTextFormat,"Translator input (Plain-text document)",
      StaticParameter(
       id='encoding',name='Encoding',description='The character encoding of the file',
       value='utf-8'
      ),
      ChoiceParameter(
       id='language',name='Language',description='The language the text is in',
       choices=[('en','English'),('nl','Dutch'),('fr','French')]
      ),
      extension='.txt',
      multi=True
    ),
    OutputTemplate('translationoutput', PlainTextFormat,
      "Esperanto translation (Plain-text document)",
      CopyMetaField('encoding','maininput.encoding')
      SetMetaField('language','eo'),
      multi=True
    )
  )
]
```

2.3.7 Parameter Conditions

It is not always possible to define all output templates straight away. Sometimes output templates are dependent on certain global parameters. For example, given a global parameter that toggles the generation of a lexicon, you want to only include the output template that describes this lexicon, if the parameter is enabled. CLAM offers a solution for such situations using the `ParameterCondition` directive.

Assume you have the following *global* parameter:

```
BooleanParameter(
  id='createlexicon',name='Create Lexicon',description='Create lexicon files',
)
```

We can then turn an output template into an output template conditional on this parameter using the following construction:

```
ParameterCondition(createlexicon=True,
  then=OutputTemplate('lexiconoutput', PlainTextFormat,
    "Lexicon (Plain-text document)",
    unique=True
  )
)
```

The first argument of ParameterCondition is the condition. Here you use the ID of the parameter and the value you want to check against. The above example illustrates an equality comparison, but other comparisons are also possible:

- *ID*=*value* – Equality; matches if the global parameter with the specified ID has the specified value.
- *ID*_equals=*value* – Same as above, the above is an alias.
- *ID*_notequals=*value* – The reverse of the above, matches if the value is *not equal*
- *ID*_lessthan=*number* – Matches if the parameter with the specified ID is less than then specified number
- *ID*_greaterthan=*number* – Matches if the parameter with the specified ID is greater than then specified number
- *ID*_lessequalthan=*number* – Matches if the parameter with the specified ID is equal or less than then specified number
- *ID*_greaterequalthan=*number* – Matches if the parameter with the specified ID is equal or greater than then specified number

After the condition you specify `then=` and optionally also `else=`, and then you specify an `OutputTemplate` or yet another `ParameterCondition`—they can be nested at will.

Parameter conditions can not only be used around output templates, but also be around metafield actors, inside the output template specification. In other words, you can make metadata fields conditional on global parameters.

2.3.8 Converters

Users do not always have their files in the format you desire as input, and asking users to convert their data may be problematic. Similarly, users may not always like the output format you offer. CLAM therefore introduces a converter framework that can do two things:

1. Convert input files from auxiliary formats to your desired format, upon upload;
2. Convert output files from your output format to secondary formats.

A converter, using the above-mentioned class names, can be included in input templates (for situation 1), and in output templates (for situation 2). Include them directly after any Parameter fields or Metafield actors.

It is important to note that the converters convert only the files themselves and not the associated metadata. This implies that these converters are intended primarily for end users and not as much for automated clients.

For most purposes, you will need to write your own converters. These are to be implemented in `clam/common/converters.py`. Some converters however will be provided out of the box. Note that the actual conversion will be performed by 3rd party software in most cases.

- `MSWordConverter` – Convert MS Word files to plain text
- `PDFConverter` – Convert PDF to plain text.
- `CharEncodingConverter` – Convert between plain text files in different character encodings.

Note that specific converters take specific parameters, consult the API reference for details.

2.3.9 Viewers

Viewers are intended for human end users, and enable visualisation of a particular file format. CLAM offers a viewer framework that enables you to write viewers for your format. Viewers may either be written within the CLAM framework, using Python, but they can also be external (non-CLAM) webservices, hosted elsewhere.

Viewers can be included in output templates. Include them directly after any metafield actors.

Development Notes
To be completed still...

2.3.10 Working with pre-installed data

Rather than letting users upload files, CLAM also offers the possibility of pre-installing input data on the server. This feature is ideally suited for dealing with data for a demo, or for offering a selection of pre-installed corpora that are too big to transfer over network. Furthermore, pre-installed data is also suited in situations where you want the user to be able to choose from several pre-installed resources, such as lexicons, grammars, etc., instead of having to upload files they may not have available.

Pre-installed data sources are called "input sources" in CLAM, not to be confused with input templates. Input sources can be specified either in an input template, or more globally.

Take a look at the following example:

```
InputTemplate('lexicon', PlainTextFormat,"Input Lexicon",
   StaticParameter(id='encoding',name='Encoding',description='Character encoding
       value='utf-8'),
   ChoiceParameter(id='language',name='Language',description='The language the t
       choices=[('en','English'),('nl','Dutch'),('fr','French')]),
   InputSource(id='lexiconA', label="Lexicon A",
    path="/path/to/lexiconA.txt",
    metadata=PlainTextFormat(None, encoding='utf-8',language='en')
   ),
   InputSource(id='lexiconB', label="Lexicon B",
    path="/path/to/lexiconB.txt",
    metadata=PlainTextFormat(None, encoding='utf-8',language='en')
   ),
   onlyinputsource=False
),
```

This defines an input template for some kind of lexicon, with two pre-defined input sources: "lexicon A" and "lexicon B". The user can choose between these, or alternatively upload a lexicon of his own. If, however, `onlyinputsource` is

set to `True`, then the user is forced to choose only from the input sources, and can't upload his own version.

Metadata can be provided either in the inputsource configuration, or by simply adding a CLAM metadata file alongside the actual file. For the file `/path/to/lexiconA.txt`, the metadata file would be `/path/to/.lexiconA.txt.METADATA` (note the initial period; metadata files are hidden).

Input sources can also be defined globally, and correspond to multiple files, i.e. they point to a directory containing multiple files instead of pointing to a single file. Let us take the example of a spelling correction demo, in which a test set consisting out of many text documents is the input source:

```
INPUTSOURCES = [
    InputSource(id='demotexts', label="Demo texts",
        path="/path/to/demotextdir/",
        metadata=PlainTextFormat(None, encoding='utf-8',language='en'),
        inputtemplate='maininput',
       ),
]
```

In these cases, it is essential to fill the `inputtemplate=` parameter. All files in the directory must be formatted according to this input template. Adding input sources for multiple input templates is done by simply defining multiple input sources.

2.4 Wrapper script

Service providers are encouraged to write a wrapper script that acts as the glue between CLAM and the NLP Application(s). CLAM will execute the wrapper script, and the wrapper script will in turn invoke the actual NLP Application(s). Using a wrapper script offers more flexibility than letting CLAM directly invoke the NLP Application, and allows the NLP Application itself to be totally independent of CLAM.

The wrapper script takes the arguments as specified in `COMMAND` in the service configuration file; see Section 2.3.3. There are some important things to take into account:

- All user-provided input has to be read from the specified input directory. A full listing of this input will be provided in the `clam.xml` data file. If you choose not to use this, but use `$PARAMETERS` instead, then you must take care that your application can identify the file formats by filename, extension or otherwise.
- All user-viewable output must be put in the specified output directory. Output files must be generated in accordance with the profiles that describe this generation.
- The wrapper should periodically output a small status message to `$STATUSFILE`. Whilst this is not mandatory, it offers valuable feedback to the user on the state of the system.
- The wrapper script is always started with the current working directory set to the selected project directory.

The wrapper script can be written in any language. Python developers will have the big advantage that they can directly tie into the CLAM Data API, which handles things such as reading the `clam.xml` data file, and make all parameters and input files (with metadata) directly accessible.

2.4.1 CLAM Data API

The key function of CLAM Data API is parsing the CLAM XML Data file that the clam webservice uses to communicate with clients. This data is parsed and all its components are made available in an instance of a CLAMData class.

Suppose your wrapper script is called with the following command definition:

```
COMMAND = "/path/to/wrapperscript.py $DATAFILE $STATUSFILE $OUTPUTDIRECTORY"
```

Your wrapper scripts then typically starts in the following fashion:

```
import sys
import clam.common.data

datafile = sys.argv[1]
```

```
statusfile = sys.argv[2]
outputdir = sys.argv[3]

clamdata = clam.common.data.getclamdata(datafile)
```

The first statements parse the command line arguments. The last statement returns a CLAMData instance, containing all data your wrapper might need.

For an extensive overview of the CLAMData class, we at this stage refer to the CLAM Data API Documentation at `http://ilk.uvt.nl/~mvgompel/clamapi/`.

Development Notes
This section will be expanded at a later stage

Chapter 3

Documentation for Service Clients

Being a RESTful webservice, automated clients can communicate with CLAM. In writing such clients, python users can benefit from the CLAM Client API, which in addition to the CLAM Data API provides a friendly high-level interface for communication with a CLAM webservice and the handling of its data. Of course clients can also be written without the Client API, and in other programming languages. For this refer to Appendix A, which provides a full specification of the RESTful API.

Whereever possible, the CLAM Client API returns a CLAMData instance as response.

Users of the CLAM Client API are best off by studying the example client provided with CLAM: `clam/clients/textstats.py`. This client is heavily commented. Moreover, an API reference can be found at `http://ilk.uvt.nl/~mvgompel/clamapi/`.

There is also a generic CLAM Client, `clam/clamclient.py`, which offers a command line interface to *any* CLAM service.

Development Notes
More elaborate documentation yet to be written..

Appendix A

RESTful specification

This appendix provides a full specification of the RESTful interface to CLAM:

URL	/
Get index of all projects	
Method	GET
Querystring	-
Response	200 - OK & CLAM XML, 401 - Unauthorised

URL	/[project]/
Get a project	
Method	GET
Querystring	-
Response	200 - OK & CLAM XML, 401 - Unauthorised, 404 - Not Found
Description	This returns the current state of the project in CLAM XML format. Depending on the state this contains a specification of all accepted parameters, all input files, and all output files. Note that errors in parameter validation are encoded in the CLAM XML response; the system will still return a 200 response.

Create new empty project	
Method	`PUT`
Querystring	-
Response	`201 - Created`, `401 - Unauthorised`, `403 - Forbidden` *(Invalid project ID)*, `403 - Forbidden` *(No project name)*
Description	This is necessary before attempting to upload any files; it initialises an empty new project.
Start a project with specified parameters	
Method	`POST`
Querystring	`inputsource`$=[inputsource_id]$ *(optional)* Other accepted parameters are defined in the Service Configuration file (and thus differs per service). The parameter ID corresponds to the parameter keys in the querystring
Response	`202 - Accepted` & CLAM XML, `401 - Unauthorised`, `404 - Not Found`, `403 - Permission Denied` & CLAM XML, `500 - Internal Server Error`
Description	This starts the running of a project, i.e. starts the actual background program with the specified service-specific parameters and provided input files. The parameters are provided in the query string; the input files are provided in separate POST requests to `/[project]/input/[filename]`, prior to this query. If any parameter errors occur or no profiles match the input files and parameters, a 403 response will be returned with errors marked in the CLAM XML. If a `500 - Server Error` is returned, then CLAM most likely is not able to invoke the underlying application. If `usecorpus=` is specified, then the designated pre-installed corpus will be used, instead of the regular input files.

Delete a project	
Method	`DELETE`
Querystring	-
Response	`200 - OK`, `401 - Unauthorised`, `404 - Not Found`
Description	Deletes a project. Any running processes will be aborted.

URL	`/[project]/input/[filename]`
Get an input file	
Method	`GET`
Querystring	-
Response	`200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`
Description	Retrieves the specified input file.
Delete an input file	
Method	`DELETE`
Querystring	-
Response	`200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`
Description	Deletes the specified input file.
Add/upload a new input file	
Method	`POST`
Querystring	`inputtemplate=`$[inputtemplate_id]$ `file=`$[HTTPfile]*$ `url=`$[download-url]*$ `contents=`$[text-content]*$ `metafile=`$[HTTPfile]$ `metadata=`$[CLAMMetadataXML]$ Other accepted parameters are defined in the various Input Templates in the Service Configuration file (and thus differs per service and input template). The parameter ID corresponds to the parameter keys in the query string.

Response	`200 - OK` & CLAM-Upload XML,`403 - Permission Denied` & CLAM-Upload XML, `401 - Unauthorised`, `404 - Not Found`
Description	This method adds a new input file. Response is returned in CLAM-Upload XML (distinct from CLAM XML!) Two arguments are mandatory: the input template, which designates what kind of file will be added and points to one of the InputTemplate IDs the webservice supports, and *one of the* query arguments marked with an asterisk. Adding a file can proceed either by uploading it from the client machine (`file`), by downloading it from another URL (`url`), or by passing the contents in the POST message itself (`contents`). Only one of these can be used at a time. Metadata can be passed in *three* different ways: 1) by simply specifying a metadata field as parameter to the querystring, with the same ID as defined in the input template. 2) setting the `metafile` attribute to a HTTP file, or 3) by setting `metadata` to the full XML string of the metadata specification.

URL	`/[project]/output/[filename]`
Get an output file	
Method	`GET`
Querystring	-
Response	`200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`
Description	Retrieves the specified output file.
Delete an output file	
Method	`DELETE`
Querystring	-
Response	`200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`
Description	Deletes the specified output file.

URL	`/[project]/output/[filename]/metadata`
Get the metadat for an output file	
Method	`GET`
Querystring	-
Response	`200 - OK` & CLAM Metadata XML, `401 - Unauthorised`, `404 - Not Found`

Description	Retrieves the metadata for the specified output file.

URL	`/[project]/input/[filename]/metadata`
Get the metadata for an input file	
Method	`GET`
Querystring	-
Response	`200 - OK` & CLAM Metadata XML, `401 - Unauthorised`, `404 - Not Found`
Description	Retrieves the metadata for the specified input file.

URL	`/[project]/output/`
Retrieve all output files as an archive	
Method	`GET`
Querystring	`format`= $zip\|tar.gz\|tar.bz2$
Response	`200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`
Delete all output files	
Method	`DELETE`
Querystring	-
Response	`200 - OK` & File contents, `401 - Unauthorised`
Description	Deletes all output files and resets the project for another run.

Appendix B

Prefined Formats

The following formats are pre-defined in CLAM. Each is a python class derived from `CLAMMetaData`, and defined in `clam/common/formats.py`:

Development Notes
To be written still...