Language and Speech Technology
Technical Report Series

Report Number LST-14-??

CLAM: Computational Linguistics Application Mediator

version 0.9.11

Documentation

Maarten van Gompel

October 28th, 2014 (pending acceptation)

Language and Speech Technology PI Group
Centre for Language Studies
Radboud University Nijmegen
P.O. Box 9103
NL-6500 HD Nijmegen
The Netherlands
http://www.ru.nl/lst

Series editors:
Nelleke Oostdijk
Antal van den Bosch
David van Leeuwen

Contents

1 Introduction **3**

1.1 Intended Audience 5

2 Documentation for Service Providers **6**

2.1 Technical details 6

2.1.1 Installation 6

2.1.2 Using CLAM with Apache 2 10

2.1.3 Using CLAM with nginx 12

2.1.4 Using CLAM with other webservers 14

2.1.5 Troubleshooting 14

2.2 Architecture 15

2.3 Beginning a new CLAM project 17

2.4 Service configuration 18

2.4.1 Server Administration 18

2.4.2 User Authentication 20

2.4.3 Command Definition 26

2.4.4 Project Paradigm: Metadata, Profiles & Parameters . . . 28

2.4.5 Parameter Specification 30
2.4.6 Profile specification 32
2.4.7 Control over filenames 37
2.4.8 Parameter Conditions 38
2.4.9 Converters 40
2.4.10 Viewers 41
2.4.11 Working with pre-installed data 41
2.4.12 Multiple profiles, identical input templates 43
2.4.13 Customising the web application 43
2.4.14 Actions 45
2.5 Wrapper script 47
2.5.1 CLAM Data API 48

3 Documentation for Service Clients **52**

A RESTful specification **54**

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 applications with a command line interface. 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. It is set up in a modular fashion and as such is easily extendible. It offers a rich API for writing clients and wrapper scripts.

The kind of applications that CLAM is originally 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. CLAM can, however, just as well be used

in fields other than NLP.

The CLAM webservice is a RESTful webservice[Fielding, 2000], meaning it uses the HTTP verbs GET, POST, PUT and DELETE to manipulate resources and returns responses using the HTTP response codes and XML. 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. The projects and all data is stored on the server.

The webservice responds in the CLAM XML format. An associated XSL stylesheet [Clark, 1999] can directly transform this to xhtml in the user's browser, thus providing a standalone web application for human 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. Morever, OAuth2 is also supported for delegating authentication*
- **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.*
- **Predefined datasets** – *Service providers may optionally predefine datasets, such as large corpora*

- **Batch Processing** – *CLAM's default project paradigm is ideally suited for batch-processing and the processing of large files. The background process may run for an undefined period*
- **Actions** – *CLAM's action paradigm is a remote procedure call mechanism in which you make available actions (any script/program or Python function) on specific URLs*.

In publication of research that makes use of this software, a citation should be given of: *"Maarten van Gompel (2014). CLAM: Computational Linguistics Application Mediator. Documentation. LST Technical Report Series 14-03."*.

CLAM is open-source software licensed under the GNU Public License v3, a copy of which can be found along with the software.

1.1 Intended Audience

CLAM and this documentation are intended for *1)* service providers; people who want to build a CLAM Webservice around their tool and/or people wanting to set up existing CLAM services on their server, and *2)* webservice users; people who want to write automated clients to communicate with CLAM webservices.

On the part of these users, a certain level of technical expertise is required and assumed, such as familiarity with UNIX/Linux systems, software development (programming) and system administration.

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.

This documentation is not intended for end-users using only the web application interface.

Chapter 2

Documentation for Service Providers

2.1 Technical details

CLAM is written in Python 2.6 [van Rossum, 2007], and is built on the webpy framework[1]. It can run stand-alone thanks to the built-in cherrypy webserver[2]; 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.

The software is designed for Unix-based systems (e.g. Linux or BSD) only. It also has been verified to run on Mac OS X, although this is less well supported and more difficult to install. Windows is not supported at all, and never will be.

2.1.1 Installation

CLAM is available from the Python Package Index; a standardised framework and repository for the installation of all kinds of Python packages. Using the Python Package Index, installation is easy on Debian/Ubuntu based Linux distributions:

[1] http://webpy.org
[2] http://cherrypy.org

```
$ sudo easy_install clam
```

This will automatically download and globally install the latest version of CLAM for you. If you already have CLAM installed and just want to update it, run the following:

```
$ sudo easy_install -U clam
```

It it wise to repeatedly issue this command every month or so, as updates are released on a regular basis.

In case `easy_install` is not yet available on your system, you first need to install it as follows on Debian/Ubuntu based distribution:

```
$ sudo apt-get install python-setuptools
```

Or for Red Hat/Fedora/CentOS:

```
# yum install python-setuptools
```

For other Linux distributions, and for Mac OS X with homebrew, fink, or macports, a similar package should exist.

After installation, CLAM is installed globally alongside all other Python packages, usually in a path such as `/usr/lib/python2.7/dist-packages`. The exact path and your Python version may differ. You can verify the availability of CLAM by opening an interactive Python interpreter and writing: "`import clam`"

If you do not have root access on your system and instead want to install CLAM locally in your home directory's python package directory, then invoke `easy_install` with the `--user` flag. This will install CLAM in a path such as `/.local/lib/python2.7/site-packages`.

```
$ easy_install --user clam
```

Alternatively, other custom paths can be specified using the `--prefix` flag, but this forces to user to manually ensure this target directory is part of the user's `$PYTHONPATH`.

Installation Details

The following software is required to run CLAM, the *easy install* process explained above should obtain and install all of these dependencies automatically, except for Python itself:

- python 2.6 (or a higher 2.x version, 2.5 may also work but is less tested. However, note that CLAM does not support 3.x yet)
- python-webpy, version 0.33 or higher
- python-lxml, version 2 or higher
- python-pycurl
- python-mysqldb (optional, needed only for MySQL support)

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`.

If you look in the directory where CLAM has been installed, the following files may be of particular interest:

- `clamservice.py` – The webservice itself, the command to be invoked to start it.
- `clamclient.py` – A very generic CLAM client, to be used from the command-line
- `clamdispatcher.py` – The default dispatcher for launching wrapper scripts
- `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, you can copy this to create your own styles.

Starting the service in stand-alone mode is done by launching `clamservice` (or `clamservice.py` directly) with the name of your service configuration. This standalone mode is intended primarily for development purposes and not recommended for production use. The below example shows how to launch the supplied "Text Statistics" demo-service:

```
$ clamservice clam.config.textstats
```

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

Git & Github

Though the Python Package Index is now the recommended way of installing and maintaining CLAM up to date, all CLAM code is hosted on github and can be cloned directly from its git repository at `http://github.com/proycon/clam`, using `git`. Cloning this CLAM repository is done as follows:

```
$ git clone git://github.com/proycon/clam.git
```

This will create a directory `clam` in your current working directory. To install CLAM globally or in your local Python packages, then you can use the included `setup.py` script. Alternatively, you may work directly from this checked out copy. The advantage of this would be that you have easier access to the sources of CLAM and can more easily modify them. Moreover, not having a global installation allows you to run different versions of CLAM concurrently, although it is still always recommended to keep your CLAM copy up to date. Using git

and github is also strongly encouraged if you are a developer seeking to improve CLAM itself, and possibly sending in patches.

Especially people migrating from earlier versions of CLAM may have already adopted this way of working. Do note that if you do not use `setup.py` then the commands `clamservice` and `clamnewproject` are only available directly as `clamservice.py` and `clamnewproject.py` from within the `clam` directory.

2.1.2 Using CLAM with Apache 2

In order to run a CLAM webservice in Apache, you have to install and configure several files. We will be using WSGI, through `mod_wsgi`, a gateway interface between Apache and Python. The following instructions assume you are at least basically familiar with Apache 2 configuration and Linux server administration in general.

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. An example script can be copied from `https://github.com/proycon/clam/blob/master/config/example.wsgi` and is shown in full below. Copy this to somewhere like `yourwebservice.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.

```
#!/usr/bin/env python
import sys
import os

#** If CLAM is not by default in your PYTHONPATH, you need
#to specify the directory that contains the
#subdirectory 'clam' here (and uncomment all lines): **
#CLAMPARENTDIR = '/path/to/clam'
#sys.path.append(CLAMPARENTDIR)
#os.environ['PYTHONPATH'] = CLAMPARENTDIR

#this is the directory that contains your service configuration file
```

```
WEBSERVICEDIR = '/path/to/yourwebservice/'
sys.path.append(WEBSERVICEDIR)
os.environ['PYTHONPATH'] = WEBSERVICEDIR #+':' + CLAMPARENTDIR

import yourwebservice #** import your configuration module here! **
import clam.clamservice
application = clam.clamservice.run_wsgi(yourwebservice)
```

3. Configure your service configuration file as explained in Section 2.4. 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. If CLAM is not installed in a default location, make sure `CLAMDIR` is properly set.

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 /yourwebservice \
 /path/to/yourwebservice/yourwebservice.wsgi/
WSGIDaemonProcess yourwebservice user=username group=groupname \
    home=/path/to/yourwebservice threads=15 maximum-requests=10000
WSGIProcessGroup yourservice
WSGIPassAuthorization On
Alias /yourwebservice/static /path/to/clam/static/
<Directory /path/to/clam/static/>
   Order deny,allow
   Allow from all
</Directory>
```

   The `WSGIScriptAlias` and `WSGIDaemonProcess` directives go on one line, but were 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. The path `/path/to/clam/static/` should be changed to where CLAM is installed and where the `static` directory is found. Depending on your installation and versions, this will be a directory like:
   `/usr/local/lib/python2.7/dist-packages/CLAM-0.9.8.3-py2.7.egg/clam/static`

5. It is always recommended to add some form of authentication or more restrictive access. You can either let CLAM handle authentication (*HTTP Digest Authentication* or *OAuth2*), in which case you need to set `WSGIPassAuthorization On`, as by default it is disabled, or you can let Apache itself handle authentication and not use CLAM's authentication mechanism.

6. Restart Apache.

Note that we run WSGI in Daemon mode using the `WSGIDaemonProcess` and `WSGIProcessGroup` directives, as opposed to embedded mode. This is the recommended way of running CLAM, and is even mandatory when using HTTP Digest Authentication. Whenever any code changes are made, simply `touch` the WSGI file (updating its modification time), and the changes will be immediately available. Embedded mode would require an apache restart when modifications are made, and it may also lead to problems with the HTTP Digest Authentication as authentication keys (nonces) may not be retainable in memory due to constant reloads. Again I'd like to emphasise that for authentication the line `WSGIPassAuthorization On` is vital, as otherwise user credentials will never each CLAM.

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 2.4. 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.

2.1.5 Troubleshooting

You may possibly encounter one of the following issues when attempting to access your CLAM service through a browser:

1. **Apache gives an Internal Server Error (HTTP 500)** – Check your Apache error log to see what happened. For additional debug output by CLAM, set `DEBUG=True` in your CLAM service configuration file.
2. **I get an empty white page** – There is probably an error in loading the XSL stylesheet that renders the web application. Please use Firefox to verify, instead of Google Chrome or Internet Explorer, as it provides more detailed error output on XSLT transformations.

3. **I get an error loading stylesheet** – The XSL stylesheet that renders the web-application can not be loaded. This is most likely due to a mismatch in URLs. The URL at which the webservice is accessed has to correspond exactly with the URL configured in the service configuration file, alternative hostnames or IPs will not work. Browsers refuse to load stylesheets from other source for security reasons. Check your settings for HOST, PORT, and URLPREFIX, and whether you accessed the service by the same URL.

4. **I get an error "No template named response"** – Check whether `CLAMDIR` is set in your service configuration file and whether it points to the directory in which CLAM resides (the directory containing `clamservice.py`)

5. **I'm using CLAM through Apache and WSGI, but authentication does not work and I am always logged in as anonymous** – Check that `WSGIPassAuthorization On` is set in your Apache configuration, and `USERS`, `USERS_MYSQL` or `OAUTH` is configured in your service configuration file.

Note that we strongly recommend developing your services using the built-in webserver, and migrating to Apache, nginx or another webserver, when deploying your final service.

2.2 Architecture

CLAM has a layered architecture, with at the core the command line 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.

CLAM presents two different paradigms for wrapping your script or application. The second is a new addition since CLAM 0.9.11 . You may use either or both at the same time.

1. **Project Paradigm** – Users create projects, upload files with optional parameters to those projects, and subsequently start the project, optionally

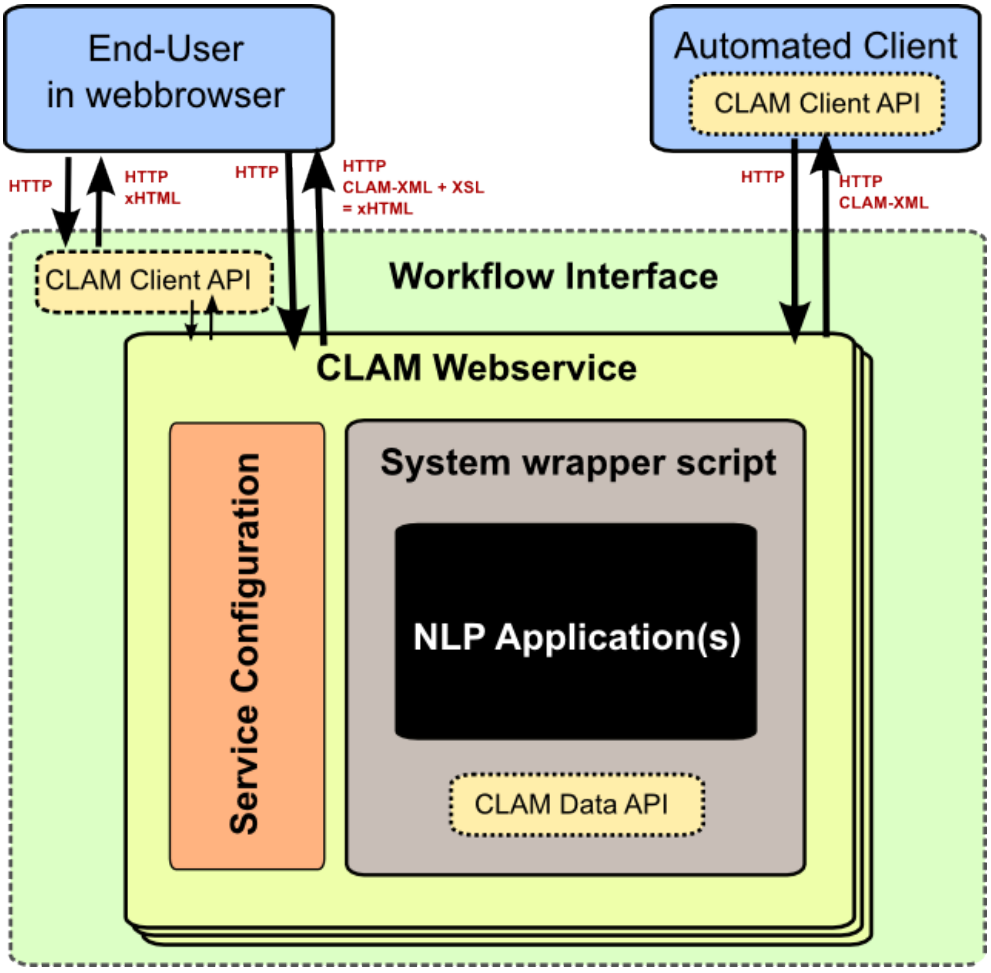

Figure 2.1: The CLAM Architecture

passing global parameters to the system. The system may run for a long time and may do batch-processing on multiple input files.

2. **Action Paradigm** – This is a more limited, and simple remote-procedure call mechanism. Users interact in real-time with the service on specific URLs, passing parameters. Unlike the project paradigm, this is not suitable for complex operations on big-data.

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

1. A service configuration file;

2. A wrapper script for your command line application;

3. A command line application (your NLP tool)

The wrapper script is not strictly mandatory if the command line 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 application unmodified. The wrapper scripts can be seen as the "glue" between CLAM and your application, taking care of any translation steps.

Note that wrapper scripts in the action paradigm are more constrained, and there may be multiple wrapper scripts for different actions.

2.3 Beginning a new CLAM project

You start a new CLAM project using the `clamnewproject` tool. It takes one argument: an identifier for your system. This identifier is for internal use and for use in filenames and may not contain any spaces or other special characters. Mind that this ID is case sensitive, so it is recommended to keep it all lower case. Example:

```
$ clamnewproject myfirstproject
```

The tool will create a directory named after the identifier, in which two template files are created which are similarly named after the chosen identifier. Both are Python scripts which you are expected to edit:

1. `myfirstproject.py` - Service Configuration FIle

2. `myfirstproject-wrapper.py` - System Wrapper Script.

These template files then need to be edited for your particular application. They are heavily commented to guide you. An `INSTRUCTIONS` file will be created in your project directory, containing instructions on what files to edit and how to start the clam service for your specific project.

You can choose not to make use of the system wrapper script and instead either write one from scratch in another language of your choice, or directly let CLAM invoke your application. Moreover, this generated wrapper script is intended for the project paradigm, not the action paradigm.

The next section will provide a detailed overview on the various ways to configure the service configuration file, and the section thereafter will deal with the system wrapper file.

2.4 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 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 `template.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.

2.4.1 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[3]. CLAM comes with a built-in webserver, which will be used when invoked directly from the command-line[4].

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.

In order to keep server load manageable, three methods are configurable in the service configuration file. First, you can set the variable `REQUIREMEMORY` to the minimum amount of free memory that has to be available (in megabytes, and not considering swap memory!). If not enough memory is free, users will not be able to launch new processes, but will receive an HTTP 500 error instead. Second, there is the `MAXLOADAVG` variable; if the 5-minute load average exceeds this number, new processes will also be rejected. Third, there is `MINDISKSPACE` and `DISK`. This sets a constraint on the minimum amount of free disk space in megabytes on the specified DISK (for example: `/dev/sda1`), which should be the disk holding `ROOT`. If any of these values is set to zero, the checks are disabled. Note though that this makes your system vulnerable to denial-of-service attacks by possibly malicious users, especially if no user authentication is configured!

Extra resource control is handled by the CLAM Dispatcher; a small program that launches and monitors your wrapper script. In your service configuration file you can configure the variable `DISPATCHER_MAXRESMEM` and `DISPATCHER_MAXTIME`. The former is the maximum memory consumption of your process, in megabytes. The latter is the maximum run-time of your process in seconds. Programs that exceed this limit will be automatically aborted. The dispatcher will check with a certain interval, configured in `DISPATCHER_POLLINTERVAL` (in seconds), if the limits have been exceeded and will take necessary action.

If you for some reason do not want to make use of the web application in CLAM, then you can disable it by setting `ENABLEWEBAPP = False`. If you want to make the webservice available at a different URL than the webapplication, then there is a small trick you can apply by setting `WEBSERVICEGHOST` to a prefix that the webservice will be made available on *without* webapplication support. If you set for example `WEBSERVICEGHOST = 'ws'` then there will be an additional "ghosted" webservice without webapplication interface running on `http://yourdomain.com/ws/`. This option is included to accommodate the wish to apply two distinct authentication schemes outside of CLAM.

[3]Most likely, the XSLT stylesheet will refuse to render the web application interface due to this mismatch

[4]unless FastCGI mode is enabled

CLAM offers a limited web-based administrative interface that allows you to view what users and projects there are, access their files, abort runs, and delete projects. This interface can be accessed on the `/admin/` URL, but requires that the logged-in user is in the list of `ADMINS` in the service configuration file. The administrative interface itself does not offer any means to adjust configuration options.

2.4.2 User Authentication

Being a RESTful webservice, user authentication proceeds over HTTP itself. CLAM implements HTTP Digest Authentication Franks et al. [1999] and OAuth2 ?. HTTP Digest Authentication, as opposed to HTTP Basic Authentication computes a hash of the username and password client-side and transmits that hash, rather than a plaintext password. User passwords are therefore only available to CLAM in hashed form. User authentication is not mandatory, but for any world-accessible environment it is most strongly recommended, for obvious security reasons.

A list of user accounts and passwords can be defined in `USERS` in the service configuration file itself. This is a simple method allowing you to quickly define users, but it is not a very scalable method. The `USERS` variable is a dictionary of usernames mapped to an md5 hash computed on the basis of the username, a string representing the security realm (by default the system ID), and the password. Projects will only be accessible and visible to their owners, unless no authentication is used at all, in which case everybody can see all projects. An example of a configuration with plain text password, converted on the fly to hashes, is found below:

```
    USERS = {
        'bob': pwhash('bob', SYSTEM_ID, 'secret'),
'alice': pwhash('alice', SYSTEM_ID, 'secret2'),
    }
```

However, computing hashes on-the-fly like in the above example is quite insecure and not recommended. You should pre-compute the hashes add those instead:

```
    USERS = {
```

```
        'bob': '6d72b6376858cf3c618c826fab1b0109',
'alice': 'e445370f57e19a8bfa454404ba3892cc',
    }
```

This pre-computation can be done in an interactive python session, executed from the CLAM directory. Make sure to change `yourconfig` in the below example to your actual service configuration file:

```
$ python
>>> from clam.common.digestauth import pwhash
>>> import clam.config.yourconfig as settings
>>> pwhash('alice', settings.SYSTEM_ID, 'secret')
'e445370f57e19a8bfa454404ba3892cc'
```

You can mark certain users as being administrators using the `ADMINS` list. Administrators can see and modify all projects.

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 2.4.6. 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.

MySQL backend

Rather than using `USERS` to define a user database in your service configuration file, a more sophisticated method is available using MySQL. The configuration variable `USERS_MYSQL` can be configured, instead of `USERS`, to point to a table in a MySQL database somewhere; the fields "username" and "password" in this table will subsequently be used to authenticate against. Custom field names are also possible. This approach allows you to use existing MySQL-based user databases. The password field is again a hashed password in the same fashion as in `USERS`, so it never contains a plaintext password. `USERS_MYSQL` is set as a Python dictionary with the following configurable keys:

```
    USERS_MYSQL = {
        'host': 'localhost',  #(default)
```

```
    'user': 'mysql_user',
    'password': 'secret_mysql_password',
    'database': 'clamopener',
    'table': 'clamusers_clamusers',
    'userfield': 'username',      #(default)
    'passwordfield': 'password',  #(default)
}
```

External authentication schemes

Extra security may also be provided on a more global webserver level, rather than in CLAM itself. For advanced service providers wanting to use external authentication schemes, such as federated identity solutions, CLAM supports the `PREAUTHHEADER` configuration directive, the value of which is a string containing an HTTP header which CLAM may read to obtain the authenticated username. This should be set by an authentication system *prior* to passing control to CLAM. An example of such a system is Shibboleth [5]. Multiple headers may be specified in `PREAUTHHEADER`, using space as delimiter, effectively creating a fallback chain. When `PREAUTHONLY` is set to `False` (default), the ultimate fallback will be CLAM's built-in user system, unless this is set to None. When such a scheme is used, proper care has to be taken to ensure that the HTTP headers can not be forged by end-users themselves! If usernames that come from external pre-authentication methods are different from those in the internal `USERS` map (if used at all), then an explicit mapping between the two may be specified in the `PREAUTHMAPPING` dictionary. Note that this pre-authentication mechanism never applies to the "ghosted" webservice, if enabled through `WEBSERVICEGHOST`. Only the regular authentication method is supported for the webservice ghost.

The below example shows an Apache configuration for a *proxy server* or *entry server* that forwards to another server on which a CLAM service runs, mediated through Shibboleth:

```
<Location /yourclamservice>
    AuthType shibboleth
    ShibRequireSession On
    ShibUseHeaders On
    require valid-user
```

[5] http://shibboleth.net

```
        ProxyPass http://realserver/yourclamservice
        ProxyPassReverse http://realserver/yourclamservice
    </Location>
```

The actual server, if it runs Apache, must always contain the directive `WSGIPassAuthorization On`.

The CLAM service configuration file can in turn be restricted to *only* accept Shibboleth authenticated users using the following settings:

```
PREAUTHHEADER = 'HTTP_EDUPERSONPRINCIPALNAME'
PREAUTHONLY = True
```

Replace `HTTP_EDUPERSONPRINCIPALNAME` with the proper HTTP header, this variable name is just an example in a CLARIN-NL context.

OAuth2

CLAM also implements OAuth2 ?, i.e. it acts as a client in the OAuth2 Authorization framework. An external OAuth2 authorization provider is responsible for authenticating you, using your user credentials to which CLAM itself will never have access. Many OAuth2 providers exists; such as Google, Facebook and Github, but you most likely want to use the OAuth2 provider of your own institution. You will need to register your webservice with your authentication provider, and obtain a `CLIENT_ID` and `CLIENT_SECRET`, the latter should be kept strictly private! These go into your service configuration file and we then enable OAuth as follows:

```
OAUTH = True
OAUTH_CLIENT_ID = "some_client_id"
OAUTH_CLIENT_SECRET = "donotsharewithanyone"
```

Note that OAuth2 by definition requires HTTPS, therefore, it can not be used with the built-in webserver but requires being embedded in a webserver such as Apache2, with SSL support.

When the user approaches the CLAM webservice, he/she will need to pass a valid access token. If none is passed, the user is instantly delegated to the OAuth2 authorization provider[6]. The authorization provider makes available a URL for authentication and for obtaining the final access token. These are configured as follows in the CLAM service configuration file:

```
OAUTH_AUTH_URL= "https://yourprovider/oauth/authenticate"
OAUTH_TOKEN_URL "https://yourprovider/oauth/token"
```

The authorization provider in turn redirects the user back to the CLAM webservice, which in turn returns the access token to the client in its XML response as follows. Note that there will just be this one tag without any children.

```
<clam xmlns:xlink="http://www.w3.org/1999/xlink" version="$version"
id="yourservice"
 name="yourservice" baseurl="https://yourservice.com/"
 oauth_access_token="1234567890">
</clam>
```

Now any subsequent call to CLAM must pass this access token, otherwise you'd simply be redirected to authenticate again. The client must thus explicitly call CLAM again. Passing the access token can be done in two ways, the recommended way is by sending the following HTTP header in your request, where the number is replaced with the actual access token:

```
Authentication: Bearer 1234567890
```

The alternative way is by passing it along with the HTTP GET/POST request. This is considered less secure as your browser may log it in its history, and the server in its access logs. It can still not be intercepted, however, as it is transmitted over HTTPS.

```
https://yourservice.com/?oauth_access_token=1234567890
```

[6]CLAM responds with a HTTP 303 - See Other

Automated clients can avoid this method, but it is necessarily used by the web-based interface. To mitigage security concerns, the access token you receive is encrypted by CLAM and bound to your IP. The passphrase for token encryption has to be configured through `OAUTH_ENCRYPTIONSECRET` in your secrive confifuration file. The web interface will furthermore explicitly ask users to log out. Logging out is done by revoking the access token with the authorization provider. For this to work, your authentication provider must offer a revoke URL, as described in RFC7009[7], which you configure in your service configuration file as follows:

```
OAUTH_REVOKE_URL = "https://yourprovider/oauth/revoke"
```

If none is set, CLAM's logout procedure will simply instruct users to clear their browser history and cache, which is clearly sub-optimal.

The only information CLAM needs from the authorization provider is a username. The setting `OAUTH_USERNAME_FUNCTION` refers to a (Python) function that obtains this from your resource provider after you have been authenticated. It gets a single argument, the instance `oauthsession` instance, and returns the username as a string. The following example shows how to implement this function for a resource provider that returns the username in JSON format. This, however, is completely provider-specific so you always have to write your own function!

```
def myprovider_username_function(oauthsession):
  r = oauthsession.get("https://yourprovider/user")
  d = json.loads(r.content)
  return d['username']

OAUTH_USERNAME_FUNCTION = myprovider_username_function
```

Various providers require the system to specify scopes, indicating the permissions the application requests from the resource provider. This can be done using the `OAUTH_SCOPE` directive in the service configuration file, which takes a list of scopes, all of which are provider-specific. The following example refers to the Google API:

[7] `https://tools.ietf.org/html/rfc7009`

```
OAUTH_SCOPE = [
     "https://www.googleapis.com/auth/userinfo.email",
     "https://www.googleapis.com/auth/userinfo.profile"
]
```

One of the problems with OAuth2 for automated clients is the authentication step that often requires user intervention. CLAM redirects unauthenticated users to the authorization provider. This is generally a website where the user enters his username and password, but the means by which authentication proceeds is not fixed by the OAuth2 specification. After authentication, the site passes a one-time authorization code back to the user, with which the user goes to CLAM to obtain the actual access token. This access token may be used for a longer time, depending on the authorization provider.

This implies that automated clients accessing the CLAM service can not authenticate in a generic fashion that is equal accross authorization providers, there is again a provider-specific component here and CLAM clients need to know how to communicate with the specific authorization provider.

At the moment, CLAM does not yet implement support for refresh tokens.

The unencrypted access token may be passed to the wrapper script if needed (has to be explicitly configured), allowing the wrapper script or underlying system to communicate with a resource provider on behalf of the user, through CLAM's client_id.

2.4.3 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.
- `$PROJECT` – The ID of the project
- `$OAUTH_ACCESS_TOKEN` – The unencrypted OAuth access token[8].

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

[8] At this is passed as a command line parameter, it may be exposed in the process list to other users on the machine CLAM is hosted on

and NLP tool are of course free to use any other locations on the filesystem for whatever other purposes.

2.4.4 Project 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 project 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[9] 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 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

[9]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.

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:

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!

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.

2.4.5 Parameter Specification

The global parameters which an NLP application, or rather the wrapper script, can take, are defined in the service configuration file. These 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[10]. Each parameter type is a Python class taking the following mandatory arguments:

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.

[10] to `common/parameters.py`

- `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).

2.4.6 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 the variable `$SEQNR` into the filename, which will be replaced by a number in sequence. After all, we cannot have multiple files with the same name. You can also use any of the metadata parameters as variable in the filename; as explained in section 2.4.7.

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.

5. `acceptarchive` – This is a boolean which can be set to True if you want to accept the upload of archives to instantly upload multiple files *for the same input template*. The file must be in zip, tar.gz or tar.bz2 format. Archives will be automatically extracted and the files within renamed according to the input template's specifications if necessary. Using this option implies that the exact same metadata will be associated with all uploaded files! This option can only be used in combination with `multi=True`. Note that archives can only be uploaded when all files therein fit the same input template!

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 append an extra extension to this inherited filename. Prior to appending an extension, you may often want to remove existing extension, you can do that with the `removeextension` attribute. 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 fictitious automatic translation system, translating to Esperanto. If an input file `x.txt` is uploaded, the output file will be named `x.translation`.

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

2.4.7 Control over filenames

There are several ways of controlling the way input and output files within a profile are named. As illustrated in the previous section, each Output Template has an Input Template as its parent, from which it inherits the filename if no explicit filename is specified. This is a very important aspect that has to be realised By default, if no `filename=`, `extension=` or `removeextension=` is specified for an Output Template, it will use the same filename as the parent Input Template. If `filename=` and `extension=` are not specified for the Input Template, then the file the user uploads will simply maintain the very same name as it is uploaded with. If `extension=` is specified, then the input file is required to have the specified extension, the web application and CLAM Client API takes care of this automatically if not the case.

In a previous section, we mentioned the use of the variable `$SEQNR` that will insert a number in when the Input Template or Output Template is in multi-mode. In addition to this, other variables can also be used. Here is an overview:

- `$SEQNR` - The sequence number of the file. Valid only if `unique=True` or `multi=False`.
- `$PROJECT` - The ID of the project.
- `$INPUTFILENAME` - The filename of the associated input file. Valid only in Output Templates.
- `$INPUTSTRIPPEDFILENAME` - The filename of the associated input file without any extensions. Valid only in Output Templates.
- `$INPUTEXTENSION` - The extension of the associated input file (without the initial period). Valid only in Output Templates.

Other than these variables pre-defined by CLAM, you can use any of the metadata parameters as variables in the filename, for input templates only. To this end, use a dollar sign followed by the ID of the parameter in the filename specification. For Output Templates, you can use metafield IDs or global parameter IDs (in that order of priority) in the same way. This syntax is valid in both `filename=` and `extension=`.

The following example illustrates a translation system that encodes the character encoding and language in the filename itself. Note also the use of the special

variable $SEQNR, which assigns a sequence number as the templates are both in multi mode.

```
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')]
      ),
      filename='input$SEQNR.$language.$encoding.txt'
      multi=True
    ),
    OutputTemplate('translationoutput', PlainTextFormat,
      "Esperanto translation (Plain-text document)",
      CopyMetaField('encoding','maininput.encoding')
      SetMetaField('language','eo'),
      filename='output$SEQNR.$language.$encoding.txt'
      multi=True
    )
  )
]
```

In addition to variables that refer to global or local parameters. There are some additional variables set by CLAM which you can use:

- $PROJECT - Is set to the project ID.
- $INPUTFILE - Is set to the project ID.

2.4.8 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 for output templates, but also for metafield actors, inside the output template specification. In other words, you can make metadata fields conditional on global parameters.

Parameter conditions can not be used for input templates, for the simple reason that in CLAM the parameters are set after the input files are uploaded. However, input templates can be *optional*, by setting `optional=True`. This means that providing such input files is optional, this also implies that any output templates that have this optional input template as a parent are also conditional on the presence of those input files.

2.4.9 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.

2.4.10 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. Several simple viewers for some formats are provided already, these are defined in `viewers.py`.

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

The below example illustrates the use of the viewer `SimpleTableViewer`, capable of showing CSV files:

```
OutputTemplate('freqlist',CSVFormat,"Frequency list",
    SimpleTableViewer(),
    SetMetaField('encoding','utf-8'),
    extension='.patterns.csv',
)
```

2.4.11 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. sources as he wants.

2.4.12 Multiple profiles, identical input templates

It is possible and sometimes necessary to define more than one profile. Recall that each profile defines what output will be generated given what input, and how the metadata is translated. Multiple profiles come into the picture as soon as you have a a disjunction of possible inputs. Imagine a spelling check system that can take either plain text as input, or a kind of XML file. In this situation you have two profiles; one for the plain-text variant, and one for the XML variant.

Now suppose there is another kind of mandatory input, a lexicon against which spell checking occurs, that is relevant for *both* profiles, and exactly the same for both profiles. In such circumstances, you could simply respecify the full input template, with the same ID as in the other profile. The most elegant solution however, is to instantiate the input template in a variable, prior to the profile definition, and then use this variable in both profiles.

2.4.13 Customising the web application

The CLAM web application offers a single uniform interface for all kinds of services. However, a certain degree of customisation is possible. One thing you may want is to include more HTML text on the pages, possibly enriched with images and hyperlinks to external sites. It is an ideal way to add extra instructions for your users. You may do so using the following variables in the service configuration file:

- `CUSTOMHTML_INDEX` - This text will be included in the index view, the overview of all projects.
- `CUSTOMHTML_PROJECTSTART` - This text will be included in the project view where the user can upload files and select parameters.
- `CUSTOMHTML_PROJECTDONE` - This text will be included in the project view when the project is done and output is ready to be viewed/downloaded.

As the HTML text will be embedded on the fly, take care *not* to include any headers, only tags that go within the HTML `body` are permitted! Always use the UTF-8 encoding and well-formed xhtml syntax.

A second kind of customisation is customisation of the style, this can be achieved by creating new CSS themes. CLAM gets shipped with the default "classic" style (which did receive a significant overhaul in CLAM 0.9). Copy, rename and adapt `style/classic.css` to create your own style. And set `STYLE` accordingly in your service configuration file.

In your service configuration file you can set a variable `INTERFACEOPTIONS`, this string is a space-separated list in which you can use the following directives to customise certain aspects of the web-interface:

- `simpleupload` – Use the simple uploader instead of the more advanced javascript-based. The simple uploader does not support multiple files but does provide full HTTP Digest Security whereas the default and more advanced uploader relies on a less sophisticated security mechanism.
- `simplepolling` – Uses a simpler polling mechanism in the stage in which CLAM awaits the completion of a process. This method simply refreshes the page periodically, whilst the default method is asynchronous but relies on a less sophisticated security mechanism.
- `secureonly` – Equals to `simpleupload` and `simplepolling`, forcing only methods that fully support HTTP Digest Authentication.
- `disablefileupload` – Disables the file uploader in the interface (do note that this is merely cosmetic and not a security mechanism, the RESTful webservice API will continue to support file uploads).
- `inputfromweb` – Enables downloading an input file from the web (do note that this is merely cosmetic and not a security mechanism, the RESTUL webservice API always supports this regardless of visibility in the interface).

- `disableliveinput` – Disables adding input through the live in-browser editor.
- `preselectinputtemplate` – Pre-select the first defined input template as default inputtemplate.

2.4.14 Actions

Since CLAM 0.9.11, a simple remote procedure call mechanism is available in addition to the more elaborate project paradigm.

This action paradigm allows you to specify *actions*, each action allows you to tie a URL to a script or Python function, and may take a number of parameters you explicitly specify. Each action is strictly independent of other actions, and complete separate of the projects, and by extension also of any files within projects and any profiles. Unlike projects, which may run over a long time period and are suited for batch processing, actions are intended for real-time communication. Typically they should return an answer in at most a couple of seconds.

Actions are specified in the service configuration file in the `ACTIONS` list. Consider the following example:

```
ACTIONS = [
  Action(id='multiply',name="Multiplier",description="Multiply two
  numbers",command="/path/to/multiply.sh $PARAMETERS",mimetype="text/plain",
  parameters=[
    IntegerParameter(id='x',name="Value 1"),
    IntegerParameter(id='y',name="Value 2"),
  ])
]
```

The ID of the action determines on what URL it listens. In this case the URL will be `/actions/multiply/`, relative to the root of your service. The name and display are for presentational purposes in the interface.

Actions will show in the web-application interface on the index page.

In this example, we specify two parameters, they will be passed *in the order they are defined* to the script. The command to be called is configured analagous to

`COMMAND`, but only a subset of the variables are supported. The most prominent is the `$PARAMETERS` variable. Note that you can set `paramflag` on the parameters to pass them with an option flag. String parameters with spaces will work without problem[11]. Actions do not have the notion of the CLAM XML datafile that wrapper scripts in the project paradigm can use, so passing command-line parameters is the only way here.

It may, however, not even be necessary to invoke an external script. Actions support calling Python functions directly. Consider the following trivial Python function for multiplication:

```
def multiply(a,b):
  return a * b
```

You can define functions in the service configuration file itself, or import it from elsewhere. We can now use this as an action directly:

```
ACTIONS = [
  Action(id='multiply',name="Multiplier",description="Multiply two
  numbers",function=multiply,mimetype="text/plain"
  parameters=[
    IntegerParameter(id='x',name="Value 1"),
    IntegerParameter(id='y',name="Value 2"),
  ])
]
```

Again, the parameters are passed in the order they are specified, irregardless of their names. A mismatch in parameters will result in an error as soon as you try to use the action. All parameters will always be validated prior to calling the script or function.

When an action completes, the standard output of the script or the return value[12] of the function is returned to the user directly (as HTTP 200) and as-is. It is therefore important to specify what MIME type the user can expect, the default is `text/plain`, but for many applications `text/html`, `text/xml` or `application/json` may be more appropriate.

[11] Do note that shells have a maximum length of all parameters combined

[12] Will be serialised to a string automatically by CLAM, assuming this is possible

By default, actions listen to both GET and POST requests. You may constrain it explicitly by specifying `method="GET"` or `method="POST"`.

When a script is called, CLAM looks at its return code to determine whether execution was successful (0). If not, CLAM will return the standard error output in a "HTTP 500 – Internal Server Error" reply. If define your own errors and return standard *output* in an HTTP 403 reply, use return code 3; for standard output in an HTTP 404 reply, use return code 4. These are just defaults, all return codes are configurable through the keyword arguments `returncodes200`, `returncodes403`, `returncodes404`, each being a list of integers.

When using Python functions, exceptions will be caught and returned to the end-user in a HTTP 500 reply (without traceback). For custom replies, Python functions may raise any instance of `web.webapi.HTTPError`.

If you enabled an authentication mechanism, as is recommended, it automatically applies to all actions. It is, however, possible to exempt certain actions from needing authentication, allowing them to serve any user anonymously. To do so, add the keyword argument `allowanonymous=True` to the configuration of the action.

If you want to use only actions and disable the project paradigm entirely, set the following in your service configuration file:

```
COMMAND = None
PROFILES = []
PARAMETERS = []
```

2.5 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 2.4.3. 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.
- Wrapper scripts often invoke the actual application using some kind of `system()` call. Take care never to pass unvalidated user-input to the shell! This makes you vulnerable for code injection attacks. The CLAM Data API offers the function `clam.common.data.shellsafe()` to help protect you.

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, makes all parameters and input files (with metadata) directly accessible, and offers a function to protect your variables against code injection when passing them to the shell.

If you used `clamnewproject` to begin your new clam service, then an example wrapper script will have been created for you, using the CLAM Data API.

2.5.1 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, which contains all data your wrapper might need, representing the state of the project and all user input.

For an extensive overview of the `CLAMData` class, we refer to the CLAM Data API Documentation at `http://packages.python.org/CLAM`. It is highly recommended to read this when your wrapper using the CLAM Data API. A few of the variables available are:

- `clamdata.system_id`
- `clamdata.project`
- `clamdata.user`
- `clamdata.status`
- `clamdata.parameters`
- `clamdata.input`

Any global parameters are available from the `clamdata` instance, by using it like a Python dictionary, where the keys correspond to the Parameter ID:

```
parameter = clamdata['parameter_id']
```

The CLAM API also has facilities to use a status file to relay progress feedback to the web-interface. Using it is as simple as importing the library and writing messages at strategic points during your program's execution:

```
import clam.common.status
clam.common.status.write(statusfile, "We are running!")
```

Progress can also be expressed through an additional completion parameter, holding a value between 0 and 1. The web-application will show a progress bar if such information is provided:

```
clam.common.status.write(statusfile, "We're half way there! Hang on!", 0.5)
```

If you have a specific input file you want to grab, you may obtain it from your `clamdata` instance by inputtemplate:

```
inputfile = clamdata.inputfile('some-inputtemplate-id')
inputfilepath = str(inputfile)
```

The variable `inputfilepath` in the above example will contain the full path to the file that was uploaded by the user for the specified input template.

Once you have a file, you can easily obtain any associated metadata in a dictionary-like fashion, for instance:

```
author = inputfile.metadata['author']
```

When you have multiple input files, you may want to iterate over all of them, the name of the inputtemplate can be obtained from the metadata:

```
for inputfile in clamdata.input:
    inputfilepath = str(inputfile)
    inputtemplate = inputfile.metadata.inputtemplate
```

The core of your wrapper script usually consists of a call to your external program. In Python this can be done through `os.system()`. Consider the following fictitious example of a program that translates an input text to the language specified by a global parameter.

```
os.system("translate -l " + clamdata['language'] + " " + \
 str(clamdata.inputfile('sourcetext')) + \
 + " > " + outputdir + "/output.txt"))
```

However, at this point you need to be conscient of possible malicious use, and make sure nobody can perform a code injection attack. The key here is to never pass unvalidated data obtained from user-input directly to the shell. CLAM's various parameters have there own validation options, the only risk left to mitigate is that of string input. If the global parameter *language* would be a free string input field, a user may insert malicious code that gets passed to the shell. To prevent this, use the `shellsafe()` function from the CLAM Data API.

```
shellsafe = clam.common.data.shellsafe #just a shortcut

os.system("translate -l " + shellsafe(clamdata['language'],"'") + " " + \
 shellsafe(str(clamdata.inputfile('sourcetext')),'"') + \
 + " > " + shellsafe(outputdir + "/output.txt") ))
```

Each variable should be wrapped in `shellsafe`, the second argument to shellsafe expresses whether to wrap the variable in quotes, and if so, which quotes. Quotes are mandatory for values containing spaces or other symbols otherwise forbidden. If no quotes are used, shellsafe does more stringent checks to prevent code injection. A Python exception is raised if the variable is not deemed safe, and the shell will not be invoked. CLAM itself will pick this up and produce an error log.

Chapter 3

Documentation for Service Clients

CLAM is designed as a RESTful webservice, which implies that usage of the four HTTP verbs (GET/POST/PUT/DELETE) on pre-defined URLs is how you can communicate with a CLAM webservice. The webservice will in turn respond with standard HTTP response codes and, where applicable, in CLAM XML format.

When writing a client for a CLAM webservice, Python users benefit greatly 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. Both are shipped as an integral part of CLAM by default. Using this API greatly facilitates writing a client for your webservice in a limited amount of time, so it is an approach to be recommended. Nevertheless, there are many valid reasons one might wish to write a client from scratch, not least as this allows you to use any programming language of your choice, or better integrate a CLAM webservice as a part of an existing application.

Appendix A of this documentation provides a full specification of the RESTful API, which will provide the technical details necessary for an implementation of a client. Moreover, each CLAM service offers an automatically tailored RESTful specification specific to the service, and example client code in Python, by pointing your browser to your service on the path `/info/`.

Users of the CLAM Client API can study the example client provided with CLAM: `clam/clients/textstats.py`. This client is heavily commented. Moreover, an API reference can be found at `http://packages.python.org/CLAM` .

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

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	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 or the server has insufficient free resources.

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	
URL	`/[project]/input/[filename]` or `/[project]/input/[inputtemplate]/[filename]`
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.

URL	`/actions/[action_id]/`
Get the metadata for an input file	
Method	`GET` and/or `POST`, may be constrained by the action
Querystring	Determined by the action
Response	`200 - OK` & Result data determined by the action, `401 - Unauthorised`, `404 - Not Found`
Description	This is a remote procedure call to run the specified action and obtain the results. The parameters are specific to the action.

If OAuth authentication is enabled and no access token is passed, almost all URLs return `HTTP 303 - See Other` and redirect to the authentication provider. At this stage, user input may be required, stopping automated clients. After the user input, or if no user input is required, the authorization provider should relay the user back to a special CLAM login page with another `HTTP 303`. This implies the client should then redo the request with the proper access token. See the section on OAuth2 authentication for more details.

Bibliography

J. Clark. XSL transformations (XSLT) version 1.0. Technical report, 11 1999. URL http://www.w3.org/TR/xslt.

R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation*. University of California, Irvine, 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

J. Franks, P. Hallam-Baker, J. Hostelter, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication (rfc2617). 1999. URL http://tools.ietf.org/html/rfc2617.

G. van Rossum. Python programming language. In *USENIX Annual Technical Conference*. USENIX, 2007. URL http://dblp.uni-trier.de/db/conf/usenix/usenix2007.html#Rossum07.