CLAM Documentation

version 0.5-pre-alpha

Maarten van Gompel
Induction of Linguistic Knowledge Research Group
Tilburg University

November 17, 2010

Contents

1 Introduction **2**

2 Documentation for Service Providers **5**

2.1 Technical details 5

2.1.1 Installation 5

2.1.2 Using CLAM with Apache 2 6

2.1.3 Using CLAM with nginx 7

2.1.4 Using CLAM with other webservers 9

2.2 Architecture 9

2.3 Service configuration 11

2.3.1 Server Administration 11

2.3.2 Command 11

2.3.3 Parameters 12

2.3.4 Inputformats & Outputformats 14

2.3.5 Users 15

2.4 Wrapper script 17

3 Documentation for Service Clients **19**

A RESTful specification **21**

Chapter 1

Introduction

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

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

CLAM is entirely written in Python and is available as open-source, it is set up in a modular fashion and as such is easily extendable.

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

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

In addition to using HTTP Error Codes for error responses, the webservice generally responds in the CLAM XML format, an associated XSL stylesheet can directly transform this to xhtml in the user's browser, thus allowing both machines and people to directly communicate with the webservice. The webservice is thus usable also as a stand-alone web-application.

This documentation is split into two parts: a section for service providers/developers and a section for service clients, users wanting to communicate with an existing service in an automated fashion.

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

Chapter 2

Documentation for Service Providers

2.1 Technical details

CLAM is written in Python 2.5, and is built on the webpy framework. It can run stand-alone thanks to the built-in cherrypy webserver. So 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).

Note that the software is designed for UNIX systems only, but any serious work in the field of NLP is UNIX based anyhow.

2.1.1 Installation

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

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

Each CLAM webservice can run standalone on any TCP port of your choice (make sure the port is open in your firewall), or you can plug it into an already existing webserver (Apache, nginx, lighttpd). For production environments, this latter option is strongly recommend.

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

- `clamservice.py` – This is the webservice itself, the command to be invoked to start it.
- `config/` – This directory contains service configuration files. Place you service configuration here.
- `config/defaults.py` – This is a default configuration template which you can copy to make your own service configuration.
- `common/` – Common Python modules for CLAM
- `common/parameters.py` – Parameter-type definitions
- `common/format.py` – Format-type definitions
- `static/style.css` – The styling for visualisation

Starting the service in standalone mode is done by simply launching clamservice.py with the name of your service configuration (test in the example below). This standalone mode is intended primarily for development purposes and not recommended for production use.

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

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

2.1.2 Using CLAM with Apache 2

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

1. Install `mod_wsgi` for Apache 2, if not already present on the system. In Debian and Ubuntu this is available as a package named `libapache2-mod-wsgi`.
2. Next we need to write a simple WSGI-script, which is a Python script that will be invoked by the webserver. Copy `clam/config/example.wsgi` to something like `clam/config/yourservice.wsgi` and adapt the script. If CLAM is not installed in a standard location where Python can find it, make sure to explicitly specify its parent directory according to the instructions in the example.
3. Configure your service configuration file as explained in Section 2.3. Take special note of Subsection where you are instructed to configure the hostname, port and optionally a URL prefix to use if the service isn't assigned a virtualhost of its own.

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

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

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

   The WSGIDaemonProcess directive goes on online, but was wrapped here for presentational purpouses. Needless to say, all paths need to be adapted according to your setup and the configuration can be extended further as desired, with for example extra authentication or more restrictive access.

5. Restart Apache

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

2.1.3 Using CLAM with nginx

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

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

```
text/xsl                              xsl;
```

2. Configure your service configuration file as explained in Section 2.3. Take special note of Subsection where you are instructed to configure the hostname, port and optionally a URL prefix to use if the service isn't 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 if course configure a `URLPREFIX` when desired, take care to update the below `location` and directives accordingly then:

   ```
   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, multiple options are possible, despite the fact we tested only these two. It should also be possible to get CLAM working on other unix based webservers, such as for example lighttpd. Although we have no CLAM-specific instructions, you may find instructions for WebPy, the framework CLAM uses, at `http://webpy.org/`, and can adapt these to CLAM.

2.2 Architecture

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

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

1. A service configuration file

2. A wrapper script for your NLP application

3. An NLP application

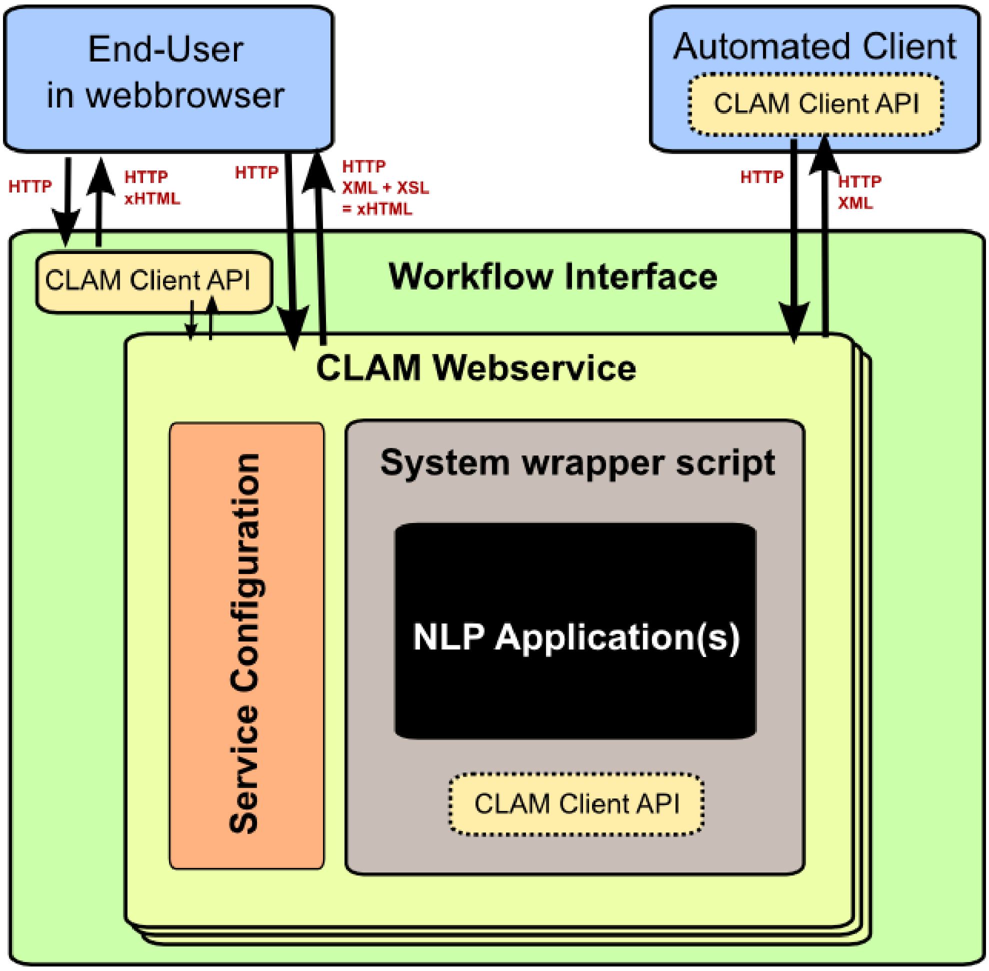

Figure 2.1: The CLAM Architecture

The wrapper script is not strictly mandatory if the NLP application can be directly invoked by CLAM. However, for more complex applications, writing a wrapper script is strongly recommended, as it offers more flexibility and better integration.

Development Notes
The CLAM Frontend, the green outer layer in Figure 2.1 does not exist yet.

2.3 Service configuration

The service configuration consists of a description of your NLP application, or rather, the wrapper script that surrounds it. It specifies what parameters the system can take, and what input and output formats are expected. The service configuration is itself a Python script. But due to its straightforward nature, knowledge of Python is not required to make your own.

The server configuration files reside in the `config/` directory. Making a new webservice starts with copying the sample `defaults.py` and editing your copy.

One of the first things to configure is the root path. All projects will be confined to the `projects/` directory under 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.

2.3.1 Server Administration

The hostname and port of the webserver can be configured in the service configuration file. Note that the hostname has to match exactly with what the end-users will use, an attempt will be made to detect this automatically if no hostname is specified. CLAM comes with a built-in webserver, which will be used when it's invoked directly from the command-line (unless FastCGI mode is enabled).

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

2.3.2 Command

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 and there are some special variables 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).
- `$PARAMETERS` - This variable will contain all parameter flags and the parameter values that have been selected by the user.
- `$DATAFILE` - This is 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. If the system wrapper script is written in Python, the CLAM Client API can be used to read this file, requiring little effort on the part of the developer.
- `$USERNAME` - The username of the logged in user.

Make sure the actual command is an absolute path or the executable is in the user's `$PATH`. Upon launch, the current working directory will be automatically set to the specific project directory. Within this directory, there will be an `input/` and `output/` directory, but use the full path as stored in `$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.

2.3.3 Parameters

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

Parameters of six types are predefined, though other types can be added to `common/parameters.py`. Each is a Python class taking the following mandatory arguments:

1. `id` – An id for internal use only.

2. `paramflag` – The parameter flag, this flag will be added to `$PARAMETERS` when the parameter is set. 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.

3. `name` – The name of this parameter, this will be shown to the user in the interface.

4. `description` – A description of this parameter, meant for the end-user.

The six parameter types are:

- `BooleanParameter` - A parameter that can only be turned on or off, represented in the interface by a checkbox. If it is turned on, the parameter flag is included in `$PARAMETERS`, if it is turned off, it is not. If `reverse=True` is set, it will do the inverse.

- `IntegerParameter` - A parameter expecting an integer number. Use `minrange=`, and `maxrange=` to restrict the range.

- `FloatParameter` - A parameter expecting a float number. Use `minrange=`, and `maxrange=` to restrict the range.

- `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`.

All parameters can take the following extra named arguments:

- `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.
- `forbid` - Set this to a list of parameter IDs. If this parameter is set, none of the others in the list may be set.
- `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.

2.3.4 Inputformats & Outputformats

CLAM takes care of the handling of input and output formats. When a user uploads a file, he chooses from a list of options to select the desired input format. Each inputformat is associated with a particular extension. CLAM automatically renames the uploaded files so they get the desired extension, relieving the end-user of this burden. If multiple extensions are defined the first is used. An extension is also associated with each outputformat, the system output files are recognised on the basis of this extension. Your system wrapper can recognise file formats also on the basis of their extension, or by explicitly getting their format from the `clam.xml` datafile. Moreover, input formats and output formats can also have associated validators, which can check if the file really is in the desired format. And for each inputformat and outputformat, a character encoding needs to be explicitly set.

The inputformats are defined in the service configuration file as a list of classes, where each class corresponds to a particular input format. The outputformats are defined in a similar way. The classes themselves are defined in `common/formats.py`, where you can add any new types (subclassed from `Format`).

Defining a format goes as follows, let's take PlainTextFormat as example:

```
INPUTFORMATS = [ PlainTextFormat('utf-8' ['.txt'], subdirectory='texts' ) ]
```

The first argument of the class constructor is the encoding, the second is a list of acceptable extensions, of which the first will be used in the rename procedure. You can redeclare instances of the same format with different encodings, but make sure the extensions you specify are unique. Some extra arguments can be provided:

- `subdirectory=` – Store all files of this format in the specific subdirectory of the input directory. This allows you to separate files of a certain format into distinct directories.

- `archivesubdirs=` – Set this to False if you want to extract archives in a flattened fashion, with disregard for any subdirectories therein.

Note that users can upload either single files in the specified format, or an archive (`zip`, `tar.gz` or `tar.bz2`) containing multiple files in the specified format.

2.3.5 Users

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

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

User authentication is not mandatory. But for any world-accessible environment it is most strongly recommended, for obvious security reasons.

The ability to view and set parameteres can be restricted to certain users, you can use the extra parameter options `allowusers=` or `denyusers=` to set this. See section 2.3.3. 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 don’t want the guest user to use.

Development Notes
Eventually, an option will probably be implemented to read users from a database. HTTPS support may also be added at a later stage.

2.4 Wrapper script

Service providers are encouraged to write a wrapper script that acts as the glue between CLAM and the NLP Application(s). CLAM will execute the wrapper script and the wrapper script will 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 selected for `COMMAND` in the service configuration file. There are some important things to take into account:

- All user-controllable input has to be read from the specified input directory. If you do not use the `clam.xml` data file, then you have to determine the file's format based on its extension. See Section 2.3.4.
- All user-viewable output must be put in the specified output directory. Make sure your system outputs files with an extension CLAM will recognise. See `$OUTPUTFORMATS` in Section 2.3.4.
- The wrapper should periodically output a small status message to `$STATUSFILE`.
- The wrapper script is always started with the current working directory set to the selected project directory.

The wrapper script can be written in any language. Python developers will have the big advantage that they can directly tie into the CLAM Client API, which handles things such as reading the `clam.xml` data file.

Development Notes
CLAM Client API will be further discussed, possibly in next chapter.

Chapter 3

Documentation for Service Clients

Being a RESTful webservice, automated clients can communicate with CLAM.

Development Notes
Yet to be written...

Appendix A

RESTful specification

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

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

URL	/[project]/
Get a project	
Method	GET
Querystring	-
Response	200 - OK & CLAM XML, 401 - Unauthorised, 404 - Not Found
Description	This returns the current state of the project in CLAM XML format, depending on the state this contains a specification of all accepted parameters, all input files, 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	`usecorpus=` $[corpus_i d]$ *(optional)* Other accepted parameters are defined in the Service Configuration file (and thus differs per service), the parameter ID corresponds to the parameter keys in the querystring
Response	`202 - Accepted` & CLAM XML, `401 - Unauthorised`, `404 - Not Found`, `403 - Permission Denied` & CLAM XML, `500 - Internal Server Error`
Description	This starts running 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 seperate 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 can't invoke the underlying application. If `usecorpus=` is specified, then the designated pre-installed corpus will be used, instead of the regular input files.

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

URL	`/[project]/input/[filename]`
Get an input file	
Method	`GET`
Querystring	-
Response	`200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`
Description	Retrieves the specified input file
Delete an input file	
Method	`DELETE`
Querystring	-
Response	`200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`
Description	Deletes the specified input file
Add/upload a new input file	
Method	`POST`
Querystring	`inputtemplate=` $[inputtemplate_id]$ `file=`$[HTTPfile]*$ `url=` $[download - url]*$ `contents=` $[text - content]$
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. Reponse is returned in CLAM-Upload XML (distinct from CLAM XML!) Two arguments are mandatory, the inputtemplate, 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.

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/
Retrieve all output files as an archive	
Method	GET
Querystring	format= $zip\vert tar.gz\vert tar.bz2$
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