# CLAM Documentation

***Release 2.4.0***

**Maarten van Gompel**

**Nov 19, 2018**

# Contents

CLAM, the Computational Linguistics Application Mediator, 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. While 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 that have 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, runs on UNIX-derived systems, and is available as open source under the GNU Public License (v3). It is set up in a modular fashion, and offers an API, and as such is easily extendable. CLAM communicates in a transparent XML format, and using XSL transformation offers a full web 2.0 web-interface for human end users.

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 *[Fielding2000]*, 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 *[XSLT]* can directly transform this to xhtml in the user's browser, thus providing a standalone web application for human end-users.

The most notable features of CLAM are:

- **RESTful webservice** – *CLAM is a fully RESTful webservice*
- **Webapplication** – *CLAM is also provides a generic web user interface for human end-users.*
- **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 and/or HTTP Basic is provided. Morever, OAuth2 is also supported for delegating authentication*
- **Metadata and provenance data** – *Extensive support for metadata and provenance data is offered*
- **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 pertaining to 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.

# CHAPTER 1

# Technical details

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

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 as well. Windows is not supported.

CHAPTER 2

# 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 3

# 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 the figure below 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 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*
2. A wrapperscript‘ 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 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.

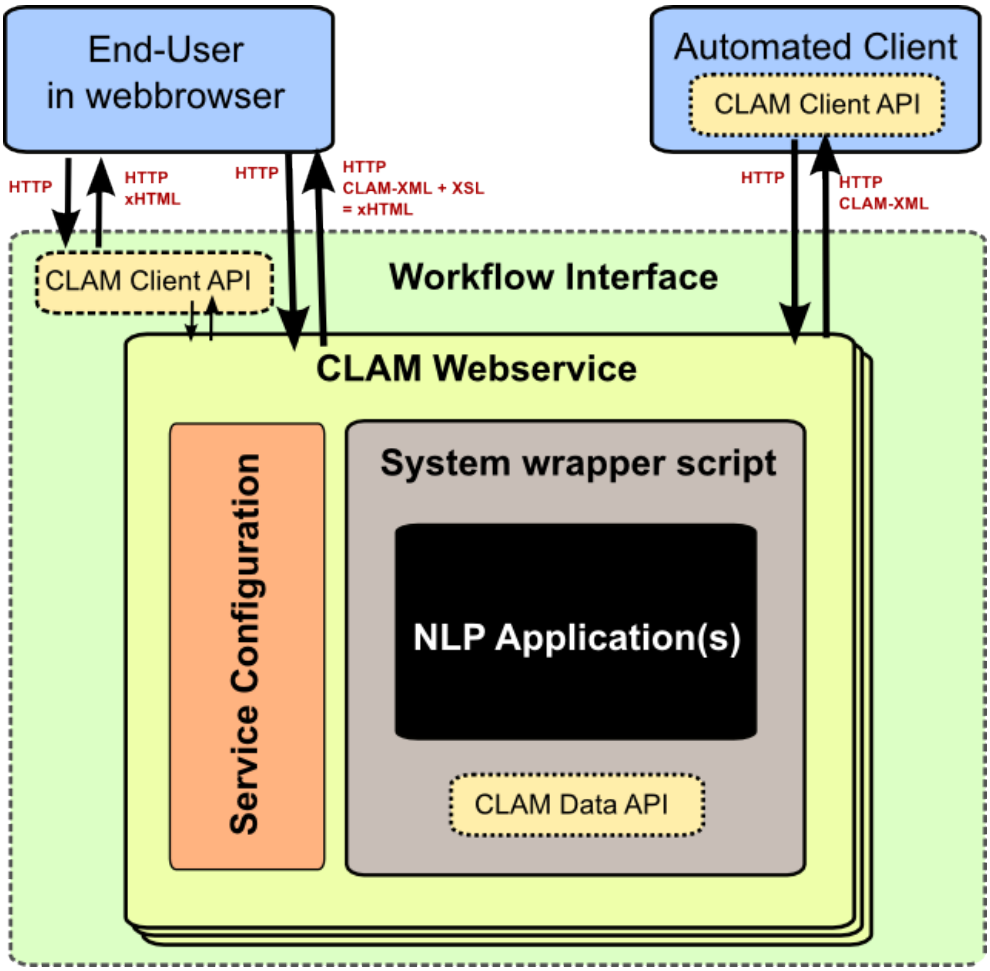


Fig. 1: The CLAM Architecture

CHAPTER 4

Table of Contents

## 4.1 Installation

### 4.1.1 Installation

CLAM is available from the Python Package Index; a standardised framework and repository for the installation of all kinds of Python packages. This is the easiest method of installing CLAM, as it will automatically fetch and install any dependencies. We recommend to use a virtual environment (`virtualenv`) if you want to install CLAM locally as a user, if you insist to install globally, prepend the following commands with `sudo`:

CLAM is written for Python 3, which we will use in this documentation. It also still offers backward compatibility with Python 2.7.

We recommend you first create a Python Virtual Environment. To create a virtual environment, which we name *clamenv* here (but you can choose any name you want), issue the following command:

```
$ virtualenv --python=python3 clamenv
```

To enter the virtual environment, type the following (note the period):

```
$ source clamenv/bin/activate.sh
```

This will change your prompt by inserting the name of the virtual environment. Now you can proceed with to install CLAM in the virtual environment:

```
$ source env/bin/activate
```

If virtualenv is not yet installed on your system, you can install it as follows (example for Debian/Ubuntu systems):

```
$ apt-get install virtualenv
```

Now you can install CLAM as follows:

```
$ pip install clam
```

If pip is not yet installed on your system, install it as follows (example for Debian/Ubuntu):

```
$ apt-get install python3-pip
```

You can verify the availability of CLAM by opening an interactive Python interpreter and writing: `import clam`

### 4.1.2 LaMachine: a meta-distribution with CLAM

We also offer LaMachine, an environment with CLAM and various CLAM webservices pre-installed, along with a lot of other NLP software. It is available as a Virtual Machine, Docker container, as well as a virtual environment through a native installation script. It is designed to facilitate installation of our software.

### 4.1.3 Installation Details

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

- python 3.3 or higher (2.7 is also still supported)
- flask
- lxml
- requests
- requests-oauthlib
- PyYAML
- mysqlclient (optional, needed only for MySQL support)
- FoLiA-Tools (optional, needed only for FoLiA 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 that you plug CLAM into a more advanced webserver (Apache, nginx, lighttpd).

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.

### 4.1.4 Usage

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

### 4.1.5 Source Code Repository

The CLAM source code is hosted on Github.

If you want to work with the latest development release of CLAM rather than the latest stable version. You can cloning this git 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 virtual environment, use the included `setup.py` script:

```
$ python3 ./setup.py install
```

Use *sudo* for global installation, or ensure you are in a virtual environment for local installation. Cloning from github directly is only recommended for people who want to contribute to CLAM development itself.

People migrating from very early versions of CLAM may have adopted a workflow that uses the clam repository from github directly, without running `setup.py`. This is no longer supported and discouraged.

## 4.2 Getting Started

Make sure you have first read the *Introduction* so you understand what CLAM is and what it's architecture is like.

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

```
$ clamnewproject myfirstproject
```

The tool will create a directory named after the identifier, in which various template files are created which are similarly named after the chosen identifier. You are expected to edit the service configuration file, a Python script, as well as a host-specific configuration file and one of the two system wrapper scripts (choose Python or Bash, or write one from scratch in your favourite language). The scripts are heavily commented to help you along, along with the documentation you are reading, this should provide you with all knowledge necessary to make a webservice.

- `myfirstproject/myfirstproject.py` - Service Configuration File
- `myfirstproject/myfirstproject.$HOSTNAME.yml` - Host-specific external configution file which is automatically included from the service configuration file if ran on the specified host. This will be addressed in *External Configuration Files*.
- `myfirstproject/myfirstproject_wrapper.py` - System Wrapper Script in Python (this is recommended over the bash version, suited for more complex webservices)

- `myfirstproject/myfirstproject_wrapper.sh` - System Wrapper Script in Bash (only suggested for simple webservices)
- `myfirstproject/myfirstproject.wsgi` - WSGI script, you probably don't need to edit this
- `setup.py` - Installation script (edit the metadata in here), run `python setup.py install` for installation in production environments or `python setup.py develop` for installation during development. (the start scripts mentioned below do this automatically for you)
- `INSTRUCTIONS.rst` - Automatically generated instructions

Moreover, some scripts and sample configurations are generated:

- `startserver_development.sh` - Start your webservice using the built-in development server
- `startserver_production.sh` - Start your webservice using the production server using uwsgi. To use this you will need to configure your webservice (e.g. Apache or nginx).
- `myfirstproject.$HOSTNAME.ini` - Uwsgi configuration (for a specific host), used for production environments
- `*.conf` - Sample configuration files for production environments using a Apache 2 or Nginx webserver. Consult the section on deployment for details.

These template files need to be edited for your particular application. They are heavily commented to guide you. The `INSTRUCTIONS.rst` file will be created in your project directory and provides instructions on what files to edit and how to start the clam service for your specific project. Starting your webservice is as easy as running `startserver_development.sh`, the script will inform you to what URL to direct your browser once the webservice is running.

You can choose not to make use of one of the generated system wrapper scripts and instead either write one from scratch in another language of your choice, or directly let CLAM invoke your application. Moreover, a wrapper is intended for the project paradigm, the action paradigm (*Actions*) does not make use of it.

### 4.2.1 Starting Your webservice

You can start your webservice in development mode with the included `startserver_development.sh` script, but not before you first read how to construct your webservice. Read the *Service configuration* documentation, and afterwards the wrapperscript documentation.

The start script simply installs your webservice and runs `clamservice` to run it, passing the module name of your webservice configuration. Make sure you first activated your Python virtual environment (if used) when calling the start script.

For production environments, read the documentation on deployment.

#### Overriding host, port and urlprefix (advanced)

The `HOST`, `PORT` and `URLPREFIX` are configured in the service configuration file or the host-specific external configuration file it includes, CLAM will attempt to automatically guess them when they are not explicitly set.

It is possible, however, to override these when launching or deploying the webserver, without changing the service configuration itself. If you use the development server, using `clamservice`, then you can pass the `-u` flag with the full URL CLAM should use. You can also set an environment variable `CLAMFORCEURL`, which has the same effect. This latter option also works when deploying CLAM through WSGI.

The most common use for this is when serving CLAM behind another reverse proxy, where automatic hostname detection could never work.

## 4.3 Service configuration

The service configuration consists of a description of your NLP application, or rather, a description of the system wrapper script that surrounds it. It specifies 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 for you to be able to make your own service configurations.

It is recommended, but not mandatory, to separate the parts of the configuration that are host-specific configuration settings from the parts that are generic. Host-specific configurations is stored in *External Configuration Files* that are dynamically included from the service configuration script. Doing so facilitates distribution and deployment on different systems late.

It is assumed you are using the `clamnewproject` tool as explain in *Getting Started*, which generates a template service configuration you can edit, including a host-specific external configuration name recognisable by the `yml` extension. When reading this section, it may help your understanding to inspect these files alongside.

One of the first things to configure is the root path (`ROOT`). All projects created in the webservice will be confined to the `projects/` directory within this root path, each project having its own subdirectory. When your underlying application or wrapper script is launched, the current working directory will be set to this project directory.

The `ROOT` directory will be automatically created upon the first run.

### 4.3.1 Server Administration

The host-specific part of the configuration contains first of all the hostname and the port where the webservice will be hosted. If not configured, automatic detection is attempted.

When CLAM runs in a production environment (see deployment) using an existing webserver without its own virtual host, it is often configured at a different URL rather than at the webserver root. In this case, the value of `URLPREFIX` should be configured accordingly. If you want your webservice to run at http://yourhost.com/yourwebservice/ for instance, then the `URLPREFIX` should be set to `yourwebservice`.

---

**Note:** In rare cases where the URL wrongly propagates to CLAM (i.e. CLAM tries to interpret your urlprefix as a project), you need to set `INTERNALURLPREFIX` to the same value. This might happen in certain WSGI set-ups, leave it unset in all other scenarios.

---

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 it will take the necessary action.

If for some reason you do not want to make use of the web-based user interface in CLAM, then you can disable it by setting `ENABLEWEBAPP = False`. Note that this is **not in any way** a security measure! Everything is technically

still as accessible. You can also disable project listing, in which case projects are only accessible if users know the exact project name. Set `LISTPROJECTS = False`.

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, and will never, offer any means to adjust service configuration options.

### 4.3.2 User Authentication

Being a RESTful webservice, user authentication proceeds over HTTP itself. CLAM implements HTTP Basic Authentication, HTTP Digest Authentication *[Franks1999]* and OAuth2 *[Hardt2012]*. HTTP Digest Authentication, contrary 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 and are not transmitted unencrypted, even over a HTTP connection. HTTP Basic Authentication, conversely, should only be use over SSL (i.e. HTTPS), and CLAM will by default disallow it if it thinks it's not running on an SSL connection.

CLAM itself does not provide SSL on the built-in development server as this is delegated to your production webserver (Apache or Nginx) instead. If you are using SSL but CLAM does not detect it, you can set `ASSUMESSL = True`. In this case HTTP Basic Authentication will be the default authentication mechanism since CLAM 2.2, but HTTP Digest Authentication is accepted too. If you're not on an SSL connection, CLAM will default to HTTP Digest Authentication only and disallow HTTP Basic Authentication.

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 and add these 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 example below to your actual service configuration file:

```
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 the documentation on *Parameter Specification*. 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.

In production environments, you will also want to set `SECRET_KEY` to a string value that is kept strictly private. It is used for cryptographically signing session data and preventing CSRF attacks (details).

### 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 forwarded authentication schemes

Authentication may also be provided on a more global webserver level, rather than in CLAM itself. An external layer takes care of the authentication and forwards a header to the actual application, i.e. CLAM. This is a feature for advanced service providers wanting to use external authentication schemes, such as federated identity solutions. IN CLAM this is implemented using the `PREAUTHHEADER` configuration directive, the value of which is a string containing the name of an HTTP header which CLAM reads 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 [4]_. Multiple headers may be specified in `PREAUTHHEADER`, using space as delimiter, effectively creating a fallback chain. If the header is not passed (which should never happen with properly configured middleware), a HTTP 401 reply will be returned.

When such a forwarded authentication scheme is used, proper care has to be taken, by the middle layer, to ensure that the HTTP headers cannot be forged by end users themselves!

It is possible that usernames that come from external pre-authentication methods are different from those in the internal `USERS` map (if used at all), an explicit mapping between the two may be specified in the `PREAUTHMAPPING` dictionary.

The example below 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
     ProxyPass http://realserver/yourclamservice
```

(continues on next page)

(continued from previous page)

```
    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 accept *only* Shibboleth authenticated users by setting `PREAUTHONLY` to `True`, as shown here:

```
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 *[Hardt2012]*, 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 (HTTP 303) to the OAuth2 authorization provider. 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 by anyone in the middle, however, as it is transmitted over HTTPS.

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

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 service configuration 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, 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 `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:

```
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.

### 4.3.3 Command Definition

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

- `$INPUTDIRECTORY` – The absolute path to the input directory where all the input files from the user will be stored (possibly in subdirectories). This input directory is the `input/` subdirectory in the project directory.
- `$OUTPUTDIRECTORY` – The absolute path to the output directory. Your system should output all of its files here, as otherwise they are not accessible through CLAM. This output directory is the `output/` subdirectory in the project directory.
- `$TMPDIRECTORY` – The absolute path to the a temporary directory. The contents of the directory will be automatically cleared as soon as your wrapper script terminates. Your system should output all of its temporary files here. This temporary directory is the `tmp/` subdirectory in the project directory.
- `$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 recommendedm however, to use $DATAFILE instead of $PARAMETERS.
- `$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 [7]_.

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

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

A profile is thus a precise specification of what output files will be produced given particular input files, and 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 input template 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.

### 4.3.5 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. 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.
- `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, 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, 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 global 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 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.
- **validator** – This should be a Python function (or other callable) taking one argument (the parameter's value), and returning either boolean indication whether the value is valid, or a (boolean, errormsg) tuple.

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).

### Parameters API

**class** clam.common.parameters.**AbstractParameter**(*id*, *name*, *description=''*, ***kwargs*)
This is the base class from which all parameter classes have to be derived.

**access**(*user*)
This method checks if the given user has access to see/set this parameter, based on the denyusers and/or allowusers option.

**allowusers = None**
You can restrict this parameter to only be available to certain users, set the usernames you want to allow here, all others are denied

**compilearg**()
This method compiles the parameter into syntax that can be used on the shell, such as for example: –param-flag=value

**constrainable**()
Should this parameter be used in checking contraints?

**denyusers = None**
You can restrict this parameter to only be available to certain users, set the usernames you want to deny access here, all others are allowed

**description = None**
A clear description for this parameter, which the user will see

**error = None**
If this parameter has any validation errors, this will be set to an error message (by default set to None, meaning no error)

**static fromxml**(*node*)
Create a Parameter instance (of any class derived from AbstractParameter!) given its XML description. Node can be a string containing XML or an lxml _Element

**id = None**
A unique alphanumeric ID

**name = None**
A representational name for this parameter, which the user will see

**paramflag = None**
The parameter flag that will be used when this parameter is passed on the commandline (using COMMAND= and $PARAMETERS) (by default set to None)

**set**(*value*)
This parameter method attempts to set a specific value for this parameter. The value will be validated first, and if it can not be set. An error message will be set in the error property of this parameter

**validate**(*value*)
Validate the parameter

**valuefrompostdata**(*postdata*)
This parameter method searches the POST data and retrieves the values it needs. It does not set the value yet though, but simply returns it. Needs to be explicitly passed to parameter.set()

**xml**(*indent=''*)
This methods renders an XML representation of this parameter, along with its selected value, and feedback on validation errors

**class** clam.common.parameters.**BooleanParameter**(*id*, *name*, *description=''*, ***kwargs*)
A parameter that takes a Boolean (True/False) value.

**compilearg**()
This method compiles the parameter into syntax that can be used on the shell, such as for example: –paramflag=value

**constrainable**()
Should this parameter be used in checking contraints?

**set**(*value=True*)
Set the boolean parameter

**unset**()

**valuefrompostdata**(*postdata*)
This parameter method searches the POST data and retrieves the values it needs. It does not set the value yet though, but simply returns it. Needs to be explicitly passed to parameter.set(). It typically returns the default None when something is left unset (but that default can be overridden)

**class** clam.common.parameters.**ChoiceParameter**(*id*, *name*, *description*, ***kwargs*)
Choice parameter, users have to choose one of the available values, or multiple values if instantiated with multi=True.

**compilearg**()
This method compiles the parameter into syntax that can be used on the shell, such as -paramflag=value

**set**(*value*)
This parameter method attempts to set a specific value for this parameter. The value will be validated first, and if it can not be set. An error message will be set in the error property of this parameter

**validate**(*values*)
Validate the parameter

**valuefrompostdata**(*postdata*)
This parameter method searches the POST data and retrieves the values it needs. It does not set the value yet though, but simply returns it. Needs to be explicitly passed to parameter.set()

**xml**(*indent=”*)
This methods renders an XML representation of this parameter, along with its selected value, and feedback on validation errors

**class** clam.common.parameters.**FloatParameter**(*id*, *name*, *description=”*, ***kwargs*)

**constrainable**()
Should this parameter be used in checking contraints?

**set**(*value*)
This parameter method attempts to set a specific value for this parameter. The value will be validated first, and if it can not be set. An error message will be set in the error property of this parameter

**validate**(*value*)
Validate the parameter

**valuefrompostdata**(*postdata*)
This parameter method searches the POST data and retrieves the values it needs. It does not set the value yet though, but simply returns it. Needs to be explicitly passed to parameter.set()

**class** clam.common.parameters.**IntegerParameter**(*id*, *name*, *description=”*, ***kwargs*)

**constrainable**()
Should this parameter be used in checking contraints?

**set**(*value*)
This parameter method attempts to set a specific value for this parameter. The value will be validated first, and if it can not be set. An error message will be set in the error property of this parameter

**validate**(*value*)
Validate the parameter

**valuefrompostdata**(*postdata*)
This parameter method searches the POST data and retrieves the values it needs. It does not set the value yet though, but simply returns it. Needs to be explicitly passed to parameter.set()

**class** clam.common.parameters.**StaticParameter**(*id*, *name*, *description=''*, ***kwargs*)
This is a parameter that can't be changed (it's a bit of a contradiction, I admit). But useful for some metadata specifications.

**class** clam.common.parameters.**StringParameter**(*id*, *name*, *description=''*, ***kwargs*)
String Parameter, taking a text value, presented as a one line input box

**compilearg**()
This method compiles the parameter into syntax that can be used on the shell, such as for example: –paramflag=value

**validate**(*value*)
Validate the parameter

**class** clam.common.parameters.**TextParameter**(*id*, *name*, *description=''*, ***kwargs*)
Text Parameter, taking a text value, presented as a multiline input box

**compilearg**()
This method compiles the parameter into syntax that can be used on the shell, such as for example: –paramflag=value

### 4.3.6 Profile specification

Multiple profiles may be specified, and all profiles are always assumed to be independent of each other. Dependencies should be together in one profile, as each profile describes how a certain type of input file is transformed into a certain type of output file. For each profile, you need to define input templates and output templates. All matching profiles are assumed to be delivered as promised. A profile matches if all input files according to the input templates of that profile are provided and if it generates output. If no input templates have been defined at all for a profile, then it will match as well, to allow for the option of producing output files that are not dependent on input files. A profile is allowed to mismatch, but if none of the profiles match, the system will produce an error, as it cannot 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 input template accepts. Formats are defined in `clam/common/formats.py`. Custom formats can be added there. Custom format classes can also be defined in the service configuration itself, after which you need to add these classes to the `CUSTOM_FORMATS` list.
3. `label` – A human readable label for the input template. This is how it will be known to users in the web application and displayed in its selection menus.

After the three mandatory arguments, you may specify any of the Parameter types to indicate the accepted/required metadata for the particular input templates. Use any of the parameter types (see *Parameter Specification*) . We will come to an example of this soon.

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. As explained in *Control over filenames*, you can also use any of the metadata parameters as variable in the filename.
4. `extension` – Files uploaded through this input template are expected to have this extension, but can have any filename. Here it does not 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 must take care to submit files 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. Uploaded archives will be automatically unpacked. It is a method to instantly upload multiple files *for the same input template*. The file must be in zip, tar.gz or tar.bz2 format. The files within the archive will be 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, illustrating of all the above:

```
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 an 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 the input templates in the profile, and the *key* part is the metadata field to copy. This allows you to combine metadata from multiple input sources 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',
```

(continues on next page)

(continued from previous page)

```
      extension='.translation',
      multi=True
    )
  )
]
```

### 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 considered when building your profiles. 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, the input file is required to have the specified extension, the web application and CLAM Client API takes care of this automatically if this is not the case.

In a previous section, we mentioned the use of the variable `$SEQNR` that will insert a number in the filename 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 pre-defined variables 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')]
      ),
```

(continues on next page)

(continued from previous page)

```
      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.

### 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 include only 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. We list them all here:

- `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 the specified number
- `ID_greaterthan=number` – Matches if the parameter with the specified ID is greater tha then specified number

- `ID_lessequalthan=number` – Matches if the parameter with the specified ID is equal or less than the specified number
- `ID_greaterequalthan=number` – Matches if the parameter with the specified ID is equal or greater than the 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 cannot 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 cannot 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.

### 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 auxiliary 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` and derived off `AbstractConverter`. 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. This converter uses the external tool catdoc by default.
- `PDFConverter` – Convert PDF to plain text. This converter uses the external tool pdftohtml by default.
- `CharEncodingConverter` – Convert between plain text files in different character encodings.

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

### 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` and derived off `AbstractViewer`.

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',
)
```

Another useful viewer is the `ForwardViewer`. It forwards the viewing request to a remote service and passes a backlink where the remote service can *download* the output file (assuming it has proper authorization!). The remote service is expected to return a HTTP 302 Redirect response which CLAM will subsequently invoke.

```
OutputTemplate('freqlist',CSVFormat,"Frequency list",
    ForwardViewer(Forwarder(id="some_remote_service",name="Some Remote Frequency List␣
↪Viewer")),
    url="https://remote.service.com/?download=$BACKLINK")),
    SetMetaField('encoding','utf-8'),
    extension='.patterns.csv',
)
```

You can also use forwarders globally to redirect all output as an archive (zip/tar.gz/tar.bz2), see *Forwarders*.

### Viewer API

**class** clam.common.viewers.**AbstractViewer**(***kwargs*)

**id = 'abstractviewer'**

**mimetype = 'text/html'**

**name = 'Unspecified Viewer'**

**view**(*file*, ***kwargs*)
Returns the view itself, in xhtml (it's recommended to use flask's template system!). file is a CLAMOutputFile instance. By default, if not overriden and a remote service is specified, this issues a GET to the remote service.

**class** clam.common.viewers.**FLATViewer**(***kwargs*)

**id = 'flatviewer'**

**name = 'Open in FLAT'**

**view**(*file*, ***kwargs*)
Returns the view itself, in xhtml (it's recommended to use flask's template system!). file is a CLAMOutputFile instance. By default, if not overriden and a remote service is specified, this issues a GET to the remote service.

**class** clam.common.viewers.**FoLiAViewer**(***kwargs*)

**id = 'foliaviewer'**

**name = 'FoLiA Viewer'**

**view**(*file*, ***kwargs*)
Returns the view itself, in xhtml (it's recommended to use flask's template system!). file is a CLAMOutputFile instance. By default, if not overriden and a remote service is specified, this issues a GET to the remote service.

**class** clam.common.viewers.**ForwardViewer**(*id*, *name*, *forwarder*, ***kwargs*)

The ForwardViewer calls a remote service and passes a backlink where the remote service can download an output file. The remote service is in turn expected to return a HTTP Redirect (302) response. It is implemented as a `Forwarder`. See *Forwarders*

**view**(*file*, ***kwargs*)

Returns the view itself, in xhtml (it's recommended to use flask's template system!). file is a CLAMOutputFile instance. By default, if not overriden and a remote service is specified, this issues a GET to the remote service.

**class** clam.common.viewers.**SimpleTableViewer**(***kwargs*)

**id = 'tableviewer'**

**name = 'Table viewer'**

**read**(*file*)

**view**(*file*, ***kwargs*)

Returns the view itself, in xhtml (it's recommended to use flask's template system!). file is a CLAMOutputFile instance. By default, if not overriden and a remote service is specified, this issues a GET to the remote service.

**class** clam.common.viewers.**SoNaRViewer**(***kwargs*)

**id = 'sonarviewer'**

**name = 'SoNaR Viewer'**

**view**(*file*, ***kwargs*)

Returns the view itself, in xhtml (it's recommended to use flask's template system!). file is a CLAMOutputFile instance. By default, if not overriden and a remote service is specified, this issues a GET to the remote service.

**class** clam.common.viewers.**XSLTViewer**(***kwargs*)

**id = 'xsltviewer'**

**name = 'XML Viewer'**

**view**(*file*, ***kwargs*)

Returns the view itself, in xhtml (it's recommended to use flask's template system!). file is a CLAMOutputFile instance. By default, if not overriden and a remote service is specified, this issues a GET to the remote service.

### Forwarders

To allow users to forward all output from one webservice to another, you can use Forwarders. The forwarder calls a remote service and passes a backlink where the remote service can *download* the output file (assuming it has proper authorization!). The remote service is expected to return a HTTP 302 Redirect response which CLAM will subsequently invoke.

```
FORWARDERS = [
       Forwarder(id="some_remote_service",name="Some Remote service",type="zip",␣
↪description="",
        url="https://remote.service.com/?downloadarchive=$BACKLINK"
```

(continues on next page)

(continued from previous page)

```
        )
]
```

**Note:**

- Forwarders can also be used as viewers for individual files. See *Viewers*
- A forwarder does *NOT* perform any upload, it just passes a download link to a service, the remote service must also have the necessary authorization to use it.

### 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 a 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 text is in',
       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 cannot 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 , the metadata file would be (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 set 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.

#### 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 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.

### 4.3.7 Customising the web interface

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.
- `CUSTOMHTML_PROJECTFAILED` - This text will be included in the project view when an error occurred while running the project

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, which 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. The `STYLE` may also refer to an absolute path of a CSS file to include.

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, while 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.

### 4.3.8 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 completely 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",
  tmpdir=False,
  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 (be ware that shells do have a maximum length for all parameters combined). 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 [13]_ 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.

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 (). If not, CLAM will return the standard error output in a "HTTP 500 – Internal Server Error" reply. If you define your own errors and return standard *output* in an HTTP 403 reply, use return code ; for standard output in an HTTP 404 reply, use return code . 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 the action invokes a script that outputs temporary files, you may set `tmpdir=True`, this will create a temporary directory for the duration of the action, which will be used as current working directory when the action runs. It will be automatically removed when the action ends. You may also explicitly pass this directory to the script you invoke with `command=` using the `$TMPDIRECTORY` variable.

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 = []
```

### 4.3.9 External Configuration Files

Since CLAM 2.3, you can define part of your webservice configuration in external YAML configuration files. In your normal service configuration file you then place a call to `loadconfig(__name__)`. This will automatically

search for external configuration files and includes any variables defined therein just as if they were defined directly. The power of this mechanism lies in the fact that it allows you to load a different external configuration file for hosts, allowing you to deploy your CLAM service on multiple hosts without changing the core of the service configuration.

The use of external configuration files is recommend and is also the default if you create new projects with `clamnewproject`.

The procedure is as follows, CLAM's `loadconfig()` function will attempt to search for a file named as follows, in the following order:

- `$CONFIGFILE` - If this environment variable is set, the exact file specified therein will be the file to load. This should be an absolute path reference rather than just a filename.
- `$SYSTEM_ID.$HOSTNAME.yml` - Here SYSTEM_ID must have been defined in the regular service configuration file, prior to calling `loadconfig()`, `$HOSTNAME` is the autodetected hostname of the system CLAM is running on.
- `$SYSTEM_ID.config.yml`
- `$HOSTNAME.yml`
- `config.yml` - Note that this filename does not contain any variable components, so it's a final catch-all solution.

CLAM will look in the following directories:

- The current working directory (so depends on how CLAM was started)
- The directory where the regular service configuration file exists

An example of a simple external configuration file in YAML syntax is:

```
root: /var/wwwdata/myservice
hostname: myhost
urlprefix: myservice
```

All field names will be automatically uppercased for CLAM (so root here becomes ROOT).

A simple form of templating is supported to refer to environment variables. Enclose the environment variable in double curly braces (no spaces).

You can define any variable, but the external configuration file is meant for host-specific configuration only; it can not be used to specify a full CLAM profile so is never a full substitute for the main service configuration file.

## 4.4 Wrapper script

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

When CLAM starts the wrapper script, it creates a `clam.xml` file containing the selection of parameters and input files provided by the user. It call the wrapper script with the arguments as specified in `COMMAND` in the serviceconf. 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`. While 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. Using the Python for your wrapper is therefore recommended.

If you used `clamnewproject` to begin your new clam service, two example wrapper scripts will have been created for you, one in Python using the CLAM Data API, and one using bash shell script. Choose one. These generated scripts are heavily commented to guide you in setting your wrapper script up. This documentation will add some further insights.

### 4.4.1 Data API

The key function of CLAM Data API is to parse 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. It is highly recommended to read the API reference for `CLAMData`. A few of the attributes available are:

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

Any global parameters set by the user 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 and . 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 with *:meth:'CLAMData.inputtemplate*:

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

The variable `inputfile` in the above example is an instance of `CLAMFile`, `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 parameters 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 aware 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 their 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 detect and produce an error log.

#### Program

A *program* (programme) describes exactly what output files will be generated on the basis of what input files. It is the concretisation of the profiles. Profiles specify how input relates to output in a generic sense, using input and output templates. The program lists what exact output files will be generates, with filenames, on the basis of exactly which input files. The program is a read-only construct generated from the profiles and the input. It is present in the CLAM XML response, the clam XML data file, and accessible to your wrapper script.

Keep in mind that this method allows you to iterate over the output files prior to their actual creation. Because it contains exact information on output and input files. It is the most elegant method to set up your wrapper script, avoiding any duplication of file names and allowing your wrapper to be set up in a filename agnostic way.

In the following example. We obtain all output files and corresponding output templates using the `CLAMData.getoutputfiles()`. For each output file, we can request the input files (and corresponding input templates) using the `CLAMData.getinputfiles()`.

Consider the following example that simply concatenates all input texts (input template `inputtext`) to a single output text (output template `outputtext`) using the unix `cat` tool:

```
for outputfile, outputtemplate in clamdata.program.getoutputfiles():
  outputfilepath = str(outputfile)
  if outputtemplate == 'outputtext':
    inputfiles_safe = ""
    for inputfile, inputtemplate in clamdata.program.getinputfiles(outputfilename):
        inputfilepath = str(inputfile)
        if inputtemplate == 'inputtext': #check is a bit obsolete in this case
            inputfiles_safe += " " + shellsafe(inputfilepath)
    if inputfiles_safe:
        os.system("cat " + inputfiles_safe + " > " + shellsafe(outputfilepath))
```

The `outputfile` and `inputfile` variables are again instances of `CLAMFile`. Their metadata parameters can be accesses through `outputfile.metadata['parameter_id']` and `inputfile.metadata['parameter_id']`.

### 4.4.2 Examples

Some example webservice configuration files and wrapper scripts are included in `clam/config` and `clam/wrappers` respectively, often similarly named.

One notable examples that are heavily commented:

- `textstats` – A simple text statistics/frequency list example for CLAM. It is a portable sample that has no external dependencies, the implementation is pure Python and done entirely in the wrapper script.

Some real-life CLAM webservice can be found in https://github.com/proycon/clamservices

### 4.4.3 Data API Reference

**class** clam.common.data.**AbstractMetaField**(*key*, *value=None*)

This abstract class is the basis for derived classes representing metadata fields of particular types. A metadata

field is in essence a (key, value) pair. These classes are used in output templates (described by the XML tag `meta`). They are not used by `CLAMMetaData`

**static fromxml**(*node*)
Static method returning an MetaField instance (any subclass of AbstractMetaField) from the given XML description. Node can be a string or an etree._Element.

**resolve**(*data*, *parameters*, *parentfile*, *relevantinputfiles*)

**xml**(*operator='set'*, *indent=''*)
Serialize the metadata field to XML

**class** clam.common.data.**Action**(**args*, ***kwargs*)

**static fromxml**(*node*)
Static method returning an Action instance from the given XML description. Node can be a string or an etree._Element.

**xml**(*indent=''*)

**exception** clam.common.data.**AuthRequired**(*msg=''*)
Raised on 401 - Authentication Required error. Service requires authentication, pass user credentials in CLAMClient constructor.

**exception** clam.common.data.**AuthenticationRequired**
This Exception is raised when authentication is required but has not been provided

**exception** clam.common.data.**BadRequest**

**class** clam.common.data.**CLAMData**(*xml*, *client=None*, *localroot=False*, *projectpath=None*, *loadmetadata=True*)
Instances of this class hold all the CLAM Data that is automatically extracted from CLAM XML responses. Its member variables are:

- `baseurl` - The base URL to the service (string)
- `projecturl` - The full URL to the selected project, if any (string)
- `status` - Can be: `clam.common.status.READY` (0),''clam.common.status.RUNNING'' (1), or `clam.common.status.DONE` (2)
- `statusmessage` - The latest status message (string)
- **`completion` - An integer between 0 and 100 indicating** the percentage towards completion.
- `parameters` - List of parameters (but use the methods instead)
- `profiles` - List of profiles (`[ Profile ]`)
- `program` - A Program instance (or None). Describes the expected outputfiles given the uploaded inputfiles. This is the concretisation of the matching profiles.
- `input` - List of input files (`[ CLAMInputFile ]`); use `inputfiles()` instead for easier access
- `output` - List of output files (`[ CLAMOutputFile ]`)
- `projects` - List of project IDs (`[ string ]`)
- `corpora` - List of pre-installed corpora
- `errors` - Boolean indicating whether there are errors in parameter specification
- `errormsg` - String containing an error message
- `oauth_access_token` - OAuth2 access token (empty if not used, string)

Note that depending on the current status of the project, not all may be available.

**baseurl = None**
String containing the base URL of the webserivice

**commandlineargs**()
Obtain a string of all parameters, using the paramater flags they were defined with, in order to pass to an external command. This is shell-safe by definition.

**corpora = None**
List of pre-installed corpora

**errormsg = None**
String containing an error message if an error occured

**errors = None**
Boolean indicating whether there are errors in parameter specification

**get**(*parameter_id*, *default=None*)

**input = None**
List of input files ([ CLAMInputFile ])

**inputfile**(*inputtemplate=None*)
Return the inputfile for the specified inputtemplate, if `inputtemplate=None`, inputfile is returned regardless of inputtemplate. This function may only return 1 and returns an error when multiple input files can be returned, use `inputfiles()` instead.

**inputfiles**(*inputtemplate=None*)
Generator yielding all inputfiles for the specified inputtemplate, if `inputtemplate=None`, inputfiles are returned regardless of inputtemplate.

**inputtemplate**(*template_id*)
Return the inputtemplate with the specified ID. This is used to resolve a inputtemplate ID to an InputTemplate object instance

**inputtemplates**()
Return all input templates as a list (of InputTemplate instances)

**loadmetadata = None**
*Automatically load metadata for input and output files? (default* – True)

**matchingprofiles**()
Generator yielding all matching profiles

**output = None**
List of output files ([ CLAMOutputFile ])

**outputtemplate**(*template_id*)
Get an output template by ID

**parameter**(*parameter_id*)
Return the specified global parameter (the entire object, not just the value)

**parametererror**()
Return the first parameter error, or False if there is none

**parameters = None**
This contains a list of (parametergroup, [parameters]) tuples.

**parseresponse**(*xml*, *localroot=False*)
Parses CLAM XML, there's usually no need to call this directly

**passparameters**()
Return all parameters as {id: value} dictionary

**profiles = None**
List of profiles ([ Profile ])

**program = None**
Program instance. Describes the expected outputfiles given the uploaded inputfiles. This is the concretisation of the matching profiles.

**projects = None**
List of projects ([ string ])

**projecturl = None**
String containing the full URL to the project, if a project was indeed selected

**status = None**
The current status of the service, returns clam.common.status.READY (1), clam.common.status.RUNNING (2), or clam.common.status.DONE (3)

**statusmessage = None**
The current status of the service in a human readable message

**class** clam.common.data.**CLAMFile**(*projectpath*, *filename*, *loadmetadata=True*, *client=None*, *requiremetadata=False*)

**attachviewers**(*profiles*)
Attach viewers *and converters* to file, automatically scan all profiles for outputtemplate or inputtemplate

**basedir = ''**

**copy**(*target*, *timeout=500*)
Copy or download this file to a new local file

**delete**()
Delete this file

**loadmetadata**()
Load metadata for this file. This is usually called automatically upon instantiation, except if explicitly disabled. Works both locally as well as for clients connecting to a CLAM service.

**metafilename**()
Returns the filename for the metadata file (not full path). Only used for local files.

**read**()
Loads all lines in memory

**readlines**()
Loads all lines in memory

**validate**()
Validate this file. Returns a boolean.

**class** clam.common.data.**CLAMInputFile**(*projectpath*, *filename*, *loadmetadata=True*, *client=None*, *requiremetadata=False*)

**basedir = 'input'**

**class** clam.common.data.**CLAMMetaData**(*file*, ***kwargs*)
A simple hash structure to hold arbitrary metadata

**allowcustomattributes = True**

**attributes = None**

**static fromxml**(*node*, *file=None*)
Read metadata from XML. Static method returning an CLAMMetaData instance (or rather; the appropriate subclass of CLAMMetaData) from the given XML description. Node can be a string or an etree._Element.

**httpheaders**()
HTTP headers to output for this format. Yields (key,value) tuples. Should be overridden in sub-classes!

**items**()
Returns all items as (key, value) tuples

**loadinlinemetadata**()
Not implemented

**mimetype = 'text/plain'**

**save**(*filename*)
Save metadata to XML file

**saveinlinemetadata**()
Not implemented

**schema = ''**

**validate**()
Validate the metadata

**xml**(*indent=''*)
Render an XML representation of the metadata

**class** clam.common.data.**CLAMOutputFile**(*projectpath*, *filename*, *loadmetadata=True*, *client=None*, *requiremetadata=False*)

**basedir = 'output'**

**class** clam.common.data.**CLAMProvenanceData**(*serviceid*, *servicename*, *serviceurl*, *outputtemplate_id*, *outputtemplate_label*, *inputfiles*, *parameters=None*, *timestamp=None*)
Holds provenance data

**static fromxml**(*node*)
Return a CLAMProvenanceData instance from the given XML description. Node can be a string or an lxml.etree._Element.

**xml**(*indent=''*)
Serialise provenance data to XML. This is included in CLAM Metadata files

**class** clam.common.data.**CMDIMetaData**(*file*, ***kwargs*)
Direct CMDI Metadata support, not implemented yet, reserved for future use

**exception** clam.common.data.**ConfigurationError**
This Exception is raised when authentication is required but has not been provided

**class** clam.common.data.**CopyMetaField**(*key*, *value=None*)
In CopyMetaField, the value is in the form of templateid.keyid, denoting where to copy from. If not keyid but only a templateid is specified, the keyid of the metafield itself will be assumed.

**resolve**(*data*, *parameters*, *parentfile*, *relevantinputfiles*)

**xml**(*indent=''*)
Serialize the metadata field to XML

**exception** clam.common.data.**FormatError**(*value*)
This Exception is raised when the CLAM response is not in the valid CLAM XML format

**class** clam.common.data.**Forwarder**(*id*, *name*, *url*, *description=''*, *type='zip'*)

**exception** clam.common.data.**HTTPError**
This Exception is raised when certain data (such a metadata), can't be retrieved over HTTP

**class** clam.common.data.**InputSource**(***kwargs*)

**check**()
Checks if this inputsource is usable in INPUTSOURCES

**isdir**()

**isfile**()

**xml**(*indent=''*)

**class** clam.common.data.**InputTemplate**(*template_id*, *formatclass*, *label*, **args*, ***kwargs*)
This class represents an input template. A slot with a certain format and function to which input files can be uploaded

**static fromxml**(*node*)
Static method returning an InputTemplate instance from the given XML description. Node can be a string or an etree._Element.

**generate**(*file*, *validatedata=None*, *inputdata=None*, *user=None*)
Convert the template into instantiated metadata, validating the data in the process and returning errors otherwise. inputdata is a dictionary-compatible structure, such as the relevant postdata. Return (success, metadata, parameters), error messages can be extracted from parameters[].error. Validatedata is a (errors,parameters) tuple that can be passed if you did validation in a prior stage, if not specified, it will be done automatically.

**json**()
Produce a JSON representation for the web interface

**match**(*metadata*, *user=None*)
Does the specified metadata match this template? returns (success,metadata,parameters)

**matchingfiles**(*projectpath*)
Checks if the input conditions are satisfied, i.e the required input files are present. We use the symbolic links .*.INPUTTEMPLATE.id.seqnr to determine this. Returns a list of matching results (seqnr, filename, inputtemplate).

**validate**(*postdata*, *user=None*)
Validate posted data against the inputtemplate

**xml**(*indent=''*)
Produce Template XML

**exception** clam.common.data.**NoConnection**

**exception** clam.common.data.**NotFound**(*msg=''*)
Raised on 404 - Not Found Errors

**class** clam.common.data.**OutputTemplate**(*template_id*, *formatclass*, *label*, **args*, ***kwargs*)

**findparent**(*inputtemplates*)
Find the most suitable parent, that is: the first matching unique/multi inputtemplate

**static fromxml**(*node*)
Static method return an OutputTemplate instance from the given XML description. Node can be a string or an etree._Element.

**generate**(*profile*, *parameters*, *projectpath*, *inputfiles*, *provenancedata=None*)
Yields (inputtemplate, inputfilename, outputfilename, metadata) tuples

**generatemetadata**(*parameters*, *parentfile*, *relevantinputfiles*, *provenancedata=None*)
Generate metadata, given a filename, parameters and a dictionary of inputdata (necessary in case we copy from it)

**getparent**(*profile*)
Resolve a parent ID

**xml**(*indent=''*)
Produce Template XML

**class** clam.common.data.**ParameterCondition**(***kwargs*)

**allpossibilities**()
Returns all possible outputtemplates that may occur (recusrively applied)

**evaluate**(*parameters*)
Returns False if there's no match, or whatever the ParameterCondition evaluates to (recursively applied!)

**static fromxml**(*node*)
Static method returning a ParameterCondition instance from the given XML description. Node can be a string or an etree._Element.

**match**(*parameters*)

**xml**(*indent=''*)

**exception** clam.common.data.**ParameterError**(*msg=''*)
Raised on Parameter Errors, i.e. when a parameter does not validate, is missing, or is otherwise set incorrectly.

**class** clam.common.data.**ParameterMetaField**(*key*, *value=None*)

**resolve**(*data*, *parameters*, *parentfile*, *relevantinputfiles*)

**xml**(*indent=''*)
Serialize the metadata field to XML

**exception** clam.common.data.**PermissionDenied**(*msg=''*)
Raised on 403 - Permission Denied Errors (but only if no CLAM XML response is provided)

**class** clam.common.data.**Profile**(**args*)

**static fromxml**(*node*)
Return a profile instance from the given XML description. Node can be a string or an etree._Element.

**generate**(*projectpath*, *parameters*, *serviceid*, *servicename*, *serviceurl*)
Generate output metadata on the basis of input files and parameters. Projectpath must be absolute. Returns a Program instance.

**match**(*projectpath*, *parameters*)
Check if the profile matches all inputdata *and* produces output given the set parameters. Returns a boolean

**matchingfiles**(*projectpath*)
Return a list of all inputfiles matching the profile (filenames)

**out**(*indent=''*)

**outputtemplates**()
Returns all outputtemplates, resolving ParameterConditions to all possibilities

**xml**(*indent=''*)
Produce XML output for the profile

**class** clam.common.data.**Program**(*projectpath*, *matchedprofiles=None*)
A Program is the concretisation of Profile. It describes the exact output files that will be created on the basis of what input files. This is in essence a dictionary structured as follows: `{outputfilename: (outputtemplate, inputfiles)}` in which `inputfiles` is a dictionary `{inputfilename: inputtemplate}`

**add**(*outputfilename*, *outputtemplate*, *inputfilename=None*, *inputtemplate=None*)
Add a new path to the program

**getinputfile**(*outputfile*, *loadmetadata=True*, *client=None*, *requiremetadata=False*)
Grabs one input file for the specified output filename (raises a KeyError exception if there is no such output, StopIteration if there are no input files for it). Shortcut for getinputfiles()

**getinputfiles**(*outputfile*, *loadmetadata=True*, *client=None*, *requiremetadata=False*)
Iterates over all input files for the specified outputfile (you may pass a CLAMOutputFile instance or a filename string). Yields (CLAMInputFile,str:inputtemplate_id) tuples. The last three arguments are passed to its constructor.

**getoutputfile**(*loadmetadata=True*, *client=None*, *requiremetadata=False*)
Grabs one output file (raises a StopIteration exception if there is none). Shortcut for getoutputfiles()

**getoutputfiles**(*loadmetadata=True*, *client=None*, *requiremetadata=False*)
Iterates over all output files and their output template. Yields (CLAMOutputFile, str:outputtemplate_id) tuples. The last three arguments are passed to its constructor.

**inputpairs**(*outputfilename*)
Iterates over all (inputfilename, inputtemplate) pairs for a specific output filename

**outputpairs**()
Iterates over all (outputfilename, outputtemplate) pairs

**update**(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

**class** clam.common.data.**RawXMLProvenanceData**(*data*)

**xml**()

**exception** clam.common.data.**ServerError**(*msg=''*)
Raised on 500 - Internal Server Error. Indicates that something went wrong on the server side.

**class** clam.common.data.**SetMetaField**(*key*, *value=None*)

**resolve**(*data*, *parameters*, *parentfile*, *relevantinputfiles*)

**xml**(*indent=''*)
Serialize the metadata field to XML

**exception** clam.common.data.**TimeOut**

**class** clam.common.data.**UnsetMetaField**(*key*, *value=None*)

**resolve**(*data*, *parameters*, *parentfile*, *relevantinputfiles*)

**xml**(*indent=''*)
Serialize the metadata field to XML

**exception** clam.common.data.**UploadError**(*msg=''*)

clam.common.data.**escape**(*s*, *quote*)

clam.common.data.**escapeshelloperators**(*s*)

clam.common.data.**getclamdata**(*filename*, *custom_formats=None*)

clam.common.data.**loadconfig**(*callername*, *required=True*)

clam.common.data.**parsexmlstring**(*node*)

clam.common.data.**processhttpcode**(*code*, *allowcodes=None*)

clam.common.data.**processparameter**(*postdata*, *parameter*, *user=None*)

clam.common.data.**processparameters**(*postdata*, *parameters*, *user=None*)

clam.common.data.**profiler**(*profiles*, *projectpath*, *parameters*, *serviceid*, *servicename*, *serviceurl*, *printdebug=None*)

Given input files and parameters, produce metadata for outputfiles. Returns a list of matched profiles (empty if none match), and a program.

clam.common.data.**resolveinputfilename**(*filename*, *parameters*, *inputtemplate*, *nextseq=0*, *project=None*)

clam.common.data.**resolveoutputfilename**(*filename*, *globalparameters*, *localparameters*, *outputtemplate*, *nextseq*, *project*, *inputfilename*)

clam.common.data.**sanitizeparameters**(*parameters*)

Construct a dictionary of parameters, for internal use only

clam.common.data.**shellsafe**(*s*, *quote=''*, *doescape=True*)

Returns the value string, wrapped in the specified quotes (if not empty), but checks and raises an Exception if the string is at risk of causing code injection

clam.common.data.**unescapeshelloperators**(*s*)

## 4.5 Deployment in production

In this section we assume you have your webservice all configured and working fine in development mode. The next step is to move it into production mode, i.e. the final deployment on a webserver of your choice.

**Warning:** Running with the built-in development server is not recommended for production as it offers sub-optimal performance, scalability, and security.

It is assumed you used the `clamnewproject` tool, as explained in *Getting Started*, to get started with your project. It generated various example configurations for production environments you can use.

Amongst the generated scripts is a WSGI script (recognisable by the `wsgi` extension). WSGI is a calling convention for web servers to call Python applications and this script provides the initial entry-point, you most likely don't need to edit it. Serving the python application is handled by uWSGI, which you can install (within your Python virtual environment) as follows:

```
$ pip install uwsgi
```

Your webservice project contains an `ini` file that provides the configuration for uwsgi to launch your webservice. You can read the uWSGI Documentation for a full understanding, but the generated template is commented and should generally be enough to get you going.

The uWSGI configuration is specific to the host you are running on so you will need to edit this `ini` file according to your server. It contains the port the uWSGI process should listen on (note that this is by definition a *different* port than the HTTP/HTTPS port you use to access your webserver!). The shell script `startserver_production.sh` in turn starts the uWSGI process with your webservice.

The next step is to forward requests from your webserver to this uWSGI process. Example configurations for nginx and Apache have been generated automatically, adapt these and include them in your webserver configuration. There are example configurations with a `URLPREFIX`, i.e. when you are not hosting the webservice at the webserver root, and without. Choose the one appropriate for your environment.

To use uWSGI with Apache, you need to install and enable the WSGI proxy module for Apache 2. On Debian/Ubuntu systems, this is installed as follows:

```
$ sudo apt-get install libapache2-mod-proxy-uwsgi
```

Apache configurations typically go into `/etc/apache2/sites-enabled`, within a `VirtualHost` context.

For nginx, uWSGI support should already be compiled in. Configurations are commonly stored in `/etc/nginx/conf.d/`. We assume the reader has sufficient experience with the webserver of his/her choice, and refer to the respective webserver's documentation for further details.

**Warning:** It is *always* recommended to add some form of authentication or more restrictive access. You can either let CLAM handle authentication (*HTTP Basic or Digest Authentication* or *OAuth2*), or you can let your webserver itself handle authentication and not use CLAM's authentication mechanism.

You will also need to configure your firewall so the port of the uwsgi process (as configured in the ini file), is *NOT* open to the public, and only the HTTP/HTTPS port is.

### 4.5.1 Alternative deployment on Apache 2 with mod_wsgi

As an alternative to using Apache with uWSGI, you can use the older `mod_wsgi` module. For this you do not need the uwsgi configuration (the `ini` file), nor the `startserver_production.sh` script.

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` for Python 2 and `libapache2-mod-wsgi-py3` for Python 3. The latter is recommended for CLAM, but you can only have one installed at the same time.
2. 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 \
  /usr/lib/python3.4/site-packages/clam-2.1-py3.4.egg/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. Make sure to adapt the static alias to where CLAM is installed and where the directory is found, this depends on your installation and versions and is subject to change on an upgrade.

3. It is always recommended to add some form of authentication or more restrictive access. You can either let CLAM handle authentication (*HTTP Basic or 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.
4. 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 using mod_wsgi, and is even mandatory when using HTTP Basic/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. 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.

### 4.5.2 Deploying CLAM with other webservers

The above configurations with Apache and Nginx are just the configurations we tested. Other webservers (such as for example lighttpd), should work too.

**See also:**

For configuration of authentication, see *User Authentication*.

## 4.6 Clients

CLAM is designed as a RESTful webservice. This means a client communicates with CLAM through the four HTTP verbs (GET/POST/PUT/DELETE) on pre-defined URLs, effectively manipulating various resources. The webservice will in turn respond with standard HTTP response codes and, where applicable, a body 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 why 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.

The *RESTful API specification* provides the full 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.

There is also a generic CLAM Client, `clamclient`, which offers a command line interface to *any* CLAM service. The CLAM Client API enables users to quickly write clients to interact with CLAM webservices of any kind. It is an abstraction layer over all lower-level network communication. Consult also the CLAM Data API, as responses returned by the webservice are almost always instantiated as CLAMData objects in the client.

### 4.6.1 Client API Reference

**class** clam.common.client.**CLAMClient**(*url*, *user=None*, *password=None*, *oauth=False*, *oauth_access_token=None*, *verify=None*, *loadmetadata=False*)

**abort**(*project*)
aborts AND deletes a project (alias of delete() ):

client.abort("myprojectname")

**action**(*action_id*, ***kwargs*)
Query an action, specify the parameters for the action as keyword parameters. An optional keyword parameter method='GET' (default) or method='POST' can be set. The character set encoding of the response can be configured using the encoding keyword parameter (defaults to utf-8 by default)

**addinput**(*project*, *inputtemplate*, *contents*, ***kwargs*)
Add an input file to the CLAM service. Explictly providing the contents as a string. This is not suitable for large files as the contents are kept in memory! Use `addinputfile()` instead for large files.

project - the ID of the project you want to add the file to. inputtemplate - The input template you want to use to add this file (InputTemplate instance) contents - The contents for the file to add (string)

**Keyword Arguments**

- **filename – the filename on the server**(***) –
- **metadata – A metadata object.**(***) –
- **metafile – A metadata file**(***) –

Any other keyword arguments will be passed as metadata and matched with the input template's parameters.

Example:

```
client.addinput("myproject", "someinputtemplate", "This is a test.", filename=
↪"test.txt")
```

With metadata, assuming such metadata parameters are defined:

```
client.addinput("myproject", "someinputtemplate", "This is a test.", filename=
↪"test.txt", parameter1="blah", parameterX=3.5))
```

**addinputfile**(*project*, *inputtemplate*, *sourcefile*, ***kwargs*)
Add/upload an input file to the CLAM service. Supports proper file upload streaming.

project - the ID of the project you want to add the file to. inputtemplate - The input template you want to use to add this file (InputTemplate instance) sourcefile - The file you want to add: string containing a filename (or instance of `file`)

**Keyword arguments (optional but recommended!):**

- `filename` - the filename on the server (will be same as sourcefile if not specified)
- `metadata` - A metadata object.

- `metafile` - A metadata file (filename)

Any other keyword arguments will be passed as metadata and matched with the input template's parameters.

Example:

```
client.addinputfile("myproject", "someinputtemplate", "/path/to/local/file")
```

With metadata, assuming such metadata parameters are defined:

```
client.addinputfile("myproject", "someinputtemplate", "/path/to/local/file",␣
↪parameter1="blah", parameterX=3.5)
```

**create** (*project*)
Create a new project:

client.create("myprojectname")

**delete** (*project*)
aborts AND deletes a project:

client.delete("myprojectname")

**download** (*project*, *filename*, *targetfilename*, *loadmetadata=None*)
Download an output file

**downloadarchive** (*project*, *targetfile*, *archiveformat='zip'*)
Download all output files as a single archive:

- *targetfile* - path for the new local file to be written
- *archiveformat* - the format of the archive, can be 'zip','gz','bz2'

Example:

```
client.downloadarchive("myproject","allresults.zip","zip")
```

**get** (*project*)
Query the project status. Returns a `CLAMData` instance or raises an exception according to the returned HTTP Status code

**getinputfilename** (*inputtemplate*, *filename*)
Determine the final filename for an input file given an inputtemplate and a given filename.

Example:

```
filenameonserver = client.getinputfilename("someinputtemplate","/path/to/
↪local/file")
```

**index** ()
Get index of projects. Returns a `CLAMData` instance. Use CLAMData.projects for the index of projects.

**initauth** ()
Initialise authentication, for internal use

**initrequest** (*data=None*)

**register_custom_formats** (*custom_formats*)
custom_formats is a list of Python classes holding custom formats the webservice may use. These must be registered with the client before the client can be used.

**request** (*url=''*, *method='GET'*, *data=None*, *parse=True*, *encoding=None*)
Issue a HTTP request and parse CLAM XML response, this is a low-level function called by all of the higher-level communication methods in this class, use those instead

**start** (*project*, ***parameters*)
Start a run. `project` is the ID of the project, and `parameters` are keyword arguments for the global parameters. Returns a `CLAMData` object or raises exceptions. Note that no exceptions are raised on parameter errors, you have to check for those manually! (Use startsafe instead if want Exceptions on parameter errors):

```
response = client.start("myprojectname", parameter1="blah", parameterX=4.2)
```

**startsafe** (*project*, ***parameters*)
Start a run. `project` is the ID of the project, and `parameters` are keyword arguments for the global parameters. Returns a `CLAMData` object or raises exceptions. This version, unlike `start()`, raises Exceptions (`ParameterError`) on parameter errors.

response = client.startsafe("myprojectname", parameter1="blah", parameterX=4.2)

**upload** (*project*, *inputtemplate*, *sourcefile*, ***kwargs*)
Alias for `addinputfile()`

`clam.common.client.`**donereadingupload** (*encoder*)
Called when the uploaded file has been read

## 4.7 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 "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 sources 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 mod_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.
6. **I am using ''URLPREFIX'' but CLAM tries to interpret the prefix as a project name** - This might happen in some WSGI setups. If this happens, set `INTERNALURLPREFIX` to the same value as `URLPREFIX`. Always leave it empty in any other scenario.

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.

If you have a new issue, please use our issue tracker to check whether it has already been reported, and if not, report it yourself.

## 4.8 RESTful API specification

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

**Note:** Note that for each webservice, an auto-generated and human readable RESTful API specification is available at the `/info/` endpoint which provides a more tailored overview. This info page also presents auto-generated example code for interacting with the webservice.

### 4.8.1 Project Index

**Endpoint** `/`

**Method** `GET`

**Request Parameters** (none)

**Description** Retrieves the project index and profile specification

**Response** `200 - OK` & CLAM XML, `401 - Unauthorised`

### 4.8.2 Project Endpoint

**Endpoint** `/[project]/`

**Method** `GET`

**Request Parameters** (none)

**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.

**Method** `PUT`

**Request Parameters** (none)

**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.

**Method** `POST`

**Request Parameters** Accepted parameters are defined in the Service Configuration file (and thus differ per service). The parameter ID corresponds to the parameter keys in the request parameters

**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, CLAM most likely is not able to invoke the underlying application or the server has insufficient free resources.

**Method** `DELETE`

**Request Parameters** The parameter `abortonly` can be set to 1 if you only want to abort a running process without deleting the entire project

**Response** `200 - OK`, `401 - Unauthorised`, `404 - Not Found`

**Description** Deletes a project. Any running processes will be aborted.

### 4.8.3 Input files

**Endpoint** `/[project]/input/[filename]`

**Method** `GET`

**Request Parameters** (none)

**Response** `200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`

**Description** Retrieves the specified input file.

**Method** `DELETE`

**Request Parameters** (none)

**Response** `200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`

**Description** Deletes the specified input file.

**Endpoint** `/[project]/input/[filename]` or `/[project]/input/[inputtemplate]/[filename]`

**Method** `POST`

**Request Parameters** `inputtemplate=[inputtemplate\_id] file=[HTTP file]* url=[download-url]* contents=[text-content]* metafile=[HTTP file] metadata=[CLAM Metadata XML]` 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, which is transmitted in the `multipart/form-data` encoding along with request parameters and metadata parameters. . The 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 request parameters, with the same ID as defined in the input template. 2) setting the `metafile` attribute to an HTTP file, or 3) by setting `metadata` to the full XML string of the metadata specification.

**Endpoint** `/[project]/input/[filename]/metadata`

**Method** `GET`

**Request Parameters** (none)

**Response** `200 - OK` & CLAM Metadata XML, `401 - Unauthorised`, `404 - Not Found`

**Description** Retrieves the metadata for the specified input file.

### 4.8.4 Output Files

**Endpoint** `/[project]/output/[filename]`

**Method** `GET`

**Request Parameters** (none)

**Response** `200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`

**Description** Retrieves the specified output file.

**Method** `DELETE`

**Request Parameters** (none)

**Response** `200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`

**Description** Deletes the specified output file.

**Endpoint** `/[project]/output/[filename]/metadata`

**Method** `GET`

**Request Parameters** (none)

**Response** `200 - OK` & CLAM Metadata XML, `401 - Unauthorised`, `404 - Not Found`

**Description** Retrieves the metadata for the specified output file.

#### Archive Download

**Endpoint** `/[project]/output/`

**Method** `GET`

**Request Parameters** `format=zip|tar.gz|tar.bz2`

**Response** `200 - OK` & File contents, `401 - Unauthorised`, `404 - Not Found`

**Description** Offers a single archive, of the desired format, including all output files

**Method** `DELETE`

**Request Parameters** (none)

**Response** `200 - OK` & File contents, `401 - Unauthorised`

**Description** Deletes all output files and resets the project for another run.

### 4.8.5 Actions

**Endpoint** `/actions/[action_id]/`

**Method** `GET` and/or `POST`, may be constrained by the action

**Request Parameters** 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.

### 4.8.6 Project entry shortcut

**This is a shortcut method (available since CLAM v0.99.17) that** combines the steps of project creation, file adding and upload, in one single GET or POST request. Although more limited than the invididual calls, and less RESTful, it facilitates the job for simpler callers:

**Endpoint** `/`

**Method** `GET` or `POST`

**Request Parameters** `project=[name|new]` (mandatory), selects and if necessary creates the project with the specified name. If the value is set to *new*, a random project name will be generated. `{inputtemplate}=[contents]` – Pass file contents for the specified input templateJ (the variable name is the inputtemplate ID), this corresponds to the `contents` variable in the non-shortcut method. `{inputtemplate}_url=[url]` – Pass a url where to obtain the file for the specified input templateJ (the variable name contains the inputtemplate ID), this corresponds to the `url` variable in the non-shortcut method. `{inputtemplate}_filename=[filename]` – Sets the desired filename for the specified input template, use in combination with one of the two parameters above. Not needed when the webservice assigns a fixed filename. `start=[0|1]` – Set this parameter to 1 if you want the project to start automatically. The default is not to start automatically. Other accepted parameters are defined in the Service Configuration file (and thus differ per service). For global parameters, the parameter ID corresponds to the parameter keys in the request parameters, for parameters pertaining to a specific input template, prepend the ID of the input template and an underscore to the parameter ID (`{inputtemplate}_`).

**Response** `200 - OK` & CLAM XML, `401 - Unauthorised`, `403 - Permission denied`

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.

## 4.9 Running a test webservice

If you installed CLAM using the above method, then you can launch a clam test webservice using the development server as follows:

```
$ clamservice -H localhost -p 8080 clam.config.textstats
```

Navigate your browser to http://localhost:8080 and verify everything works

Note: It is important to regularly keep CLAM up to date as fixes and improvements are implemented on a regular basis. Update CLAM using:

```
$ pip install -U clam
```

or if you used easy_install:

```
$ easy_install -U clam
```

## 4.10 Installing a particular clam webservice for production use

When installating a particular CLAM webservice on a new server, it is first necessary to edit the service configuration file of the webservice and make sure all the paths in there are set correctly for the new server. Of interest is in particular the ROOT path, which is where user data will be stored, this directory must exist and be writable by the webserver.

For testing, the built-in development server can be used. Suppose the webservice configuration is in /path/to/mywebservice/ and is called mywebservice.py, then the development server can be started as follows:

```
$ clamservice -P /path/to/mywebservice mywebservice
```

For production, however, it is strongly recommended to embed CLAM in Apache or nginx. This is the typically task of a system administrator, as certain skills are necessary and assumed. All this is explained in detail in the CLAM Manual, obtainable from https://proycon.github.io/clam/ .

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Bibliography


[Fielding2000] 18. (t) Fielding (2000). Architectural Styles and the DEsign of Network-based Software Architecture. Doctoral Dissertation. University of California, Irvine. (HTML)

[XSLT] 10. Clark (1999). XSL Transformations (XSLT) Version 1.0. W3C Recommendation. http://www.w3.org/TR/xslt

[python] Python Software Foundation. Python Language Reference. Available at https://www.python.org

[flask] http://flask.pocoo.org

[Franks1999] 10. Franks, P. Hallam-Baker, J. Hostelter, S. Lawrence, P.Leach, A. Luotonen and L. Stewart (1999). HTTP Authentication: Basic and Digest Access Authentication (RFC2617). The Internet Engineering Task Force (IETF). (HTML)

[Hardt2012] 4. Hardt (2012) The OAuth 2.0 Authorization Framework (RFC6749). **‘(Text) <http://www.rfc-editor.org/rfc/rfc6749.txt‘_**

# Python Module Index

## C

# Index

## J



## L



## M



## N



## O



## P



## R



## S

## T



## U



## V



## X