CLAM Documentation

Release 2.2.1

Maarten van Gompel

Oct 25, 2017

Contents

CLAM allows you to quickly and transparently transform your Natural Language Processing application into a RESTful webservice, with which both human end-users as well as automated clients can interact. CLAM takes a description of your system and wraps itself around the system, allowing end-users or automated clients 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 once it is completed.

CLAM is set up in a universal fashion, requiring minimal effort on the part of the service developer. Your actual NLP application is treated as a black box, of which only the parameters, input formats and output formats need to be described. Your application itself needs 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.

This documentation only concerns the API. F For *full documentation* consult the CLAM manual, also accessible through the CLAM website at http://proycon.github.io/clam . It is recommended to read this prior to starting with this API documentation.

Contents:

CHAPTER 1

CLAM Client API

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.

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

CHAPTER 2

CLAM Formats

class clam.common.converters.**AbstractConverter**(*id*, ***kwargs*)

acceptforinput = []

acceptforoutput = []

convertforinput(*filepath*, *metadata*)
Convert from target format into one of the source formats. Relevant if converters are used in InputTemplates. Metadata already is metadata for the to-be-generated file. 'filepath' is both the source and the target file, the source file will be erased and overwritten with the conversion result!

convertforoutput(*outputfile*)
Convert from one of the source formats into target format. Relevant if converters are used in OutputTemplates. Sourcefile is a CLAMOutputFile instance.

label = '(ERROR: label not overriden from AbstractConverter!)'

class clam.common.converters.**CharEncodingConverter**(*id*, ***kwargs*)

acceptforinput = [<class 'clam.common.formats.PlainTextFormat'>]

acceptforoutput = [<class 'clam.common.formats.PlainTextFormat'>]

convertforinput(*filepath*, *metadata=None*)
Convert from target format into one of the source formats. Relevant if converters are used in InputTemplates. Metadata already is metadata for the to-be-generated file.

convertforoutput(*outputfile*)
Convert from one of the source formats into target format. Relevant if converters are used in OutputTemplates. Outputfile is a CLAMOutputFile instance.

label = 'CharEncodingConverter'

class clam.common.converters.**MSWordConverter**(*id*, ***kwargs*)

acceptforinput = [<class 'clam.common.formats.PlainTextFormat'>]

convertforinput(*filepath*, *metadata=None*)

converttool = 'catdoc'

class clam.common.converters.**PDFtoHTMLConverter**(*id*, ***kwargs*)

acceptforinput = [<class 'clam.common.formats.HTMLFormat'>]

convertforinput(*filepath*, *metadata=None*)

converttool = 'pdftohtml'

class clam.common.converters.**PDFtoTextConverter**(*id*, ***kwargs*)

acceptforinput = [<class 'clam.common.formats.PlainTextFormat'>]

convertforinput(*filepath*, *metadata=None*)

converttool = 'pdftotext'

CHAPTER 3

CLAM Data API

The CLAM Data API is at the heart of CLAM. It contains various data structures CLAM uses, such as the Profiles, Input Templates, Output Templates, Metadata, etc... This API is used by CLAM internally but is also designed to be used in your system wrapper scripts and clients!

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

`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

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=''*)

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

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=''*)

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(*src*)

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=''*)

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

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

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

xml(*indent=''*)

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.**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*)

CHAPTER 4

CLAM Formats

class clam.common.formats.**AlpinoXMLFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'text/xml'

name = 'Alpino XML'

scheme = ''

class clam.common.formats.**BinaryDataFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'application/octet-stream'

name = 'Application-specific Binary Data'

class clam.common.formats.**CSVFormat**(*file*, ***kwargs*)

attributes = {'encoding': True, 'language': False}

mimetype = 'text/csv'

name = 'Comma separated file'

class clam.common.formats.**DCOIFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'text/xml'

name = 'DCOI format'

scheme = ''

class clam.common.formats.**DjVuFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'image/x-djvu'

name = 'DjVu format'

class clam.common.formats.**ExampleFormat**(*file*, ***kwargs*)
This is an Example format, please inspect its source code if you want to create custom formats!

allowcustomattributes = True

httpheaders()
HTTP headers to output for this format. Yields (key,value) tuples.

loadinlinemetadata()
If there is metadata IN the actual file, this method should extract it and assign it to this object. Will be automatically called from constructor. Note that the file (CLAMFile) is accessible through self.file

mimetype = 'text/plain'

saveinlinemetadata()
If there is metadata that should be IN the actual file, this method can store it. Note that the file (CLAMFile) is accessible through self.file

scheme = None

validate()
Add your validation method here, should return True or False

class clam.common.formats.**FoLiAXMLFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'text/xml'

name = 'FoLiA XML'

scheme = ''

class clam.common.formats.**GifImageFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'image/gif'

name = 'Gif Image'

class clam.common.formats.**HTMLFormat**(*file*, ***kwargs*)
HTML Format Definition. This format has one required attribute: encoding

attributes = {'encoding': True, 'language': False}

httpheaders()
HTTP headers to output for this format. Yields (key,value) tuples.

mimetype = 'text/html'

class clam.common.formats.**JpegImageFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'image/jpeg'

name = 'Jpeg Image'

class clam.common.formats.**KBXMLFormat**(*file*, ***kwargs*)

mimetype = 'text/xml'

name = 'Koninklijke Bibliotheek XML-formaat'

scheme = ''

class clam.common.formats.**MP3AudioFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'audio/mpeg'

name = 'MP3 Audio File'

class clam.common.formats.**MSWordFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'application/msword'

name = 'Microsoft Word format'

scheme = ''

class clam.common.formats.**MpegVideoFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'video/mpeg'

name = 'Mpeg Video'

class clam.common.formats.**OggAudioFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'audio/ogg'

name = 'Ogg Audio File'

class clam.common.formats.**OggVideoFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'audio/ogg'

name = 'Ogg Video File'

class clam.common.formats.**OpenDocumentTextFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'application/vnd.oasis.opendocument.text'

name = 'Open Document Text Format'

class clam.common.formats.**PDFFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'application/pdf'

name = 'PDF'

class clam.common.formats.**PlainTextFormat**(*file*, ***kwargs*)

Plain Text Format Definition. This format has one required attribute: encoding

attributes = {'encoding': True, 'language': False}

httpheaders()

HTTP headers to output for this format. Yields (key,value) tuples.

mimetype = 'text/plain'

class clam.common.formats.**PngImageFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'image/png'

name = 'PNG Image'

class clam.common.formats.**TICCLShadowOutputXML**(*file*, ***kwargs*)

mimetype = 'text/xml'

name = 'Ticcl Shadow Output'

scheme = ''

class clam.common.formats.**TICCLVariantOutputXML**(*file*, ***kwargs*)

mimetype = 'text/xml'

name = 'Ticcl Variant Output'

scheme = ''

class clam.common.formats.**TadpoleFormat**(*file*, ***kwargs*)

attributes = {'morphologicalanalysis': ['yes', 'no'], 'tokenisation': 'yes', 'parsing': ['yes', 'no'], 'mwudetection':

mimetype = 'text/plain'

name = 'Tadpole Columned Output Format'

class clam.common.formats.**TiffImageFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'image/tiff'

name = 'Tiff Image'

class clam.common.formats.**UndefinedXMLFormat**(*file*, ***kwargs*)

mimetype = 'text/xml'

name = 'Undefined XML Format'

scheme = ''

class clam.common.formats.**WaveAudioFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'audio/wav'

name = 'Wave Audio File'

class clam.common.formats.**XMLStyleSheet**(*file*, ***kwargs*)

attributes = {}

mimetype = 'application/xslt+xml'

name = 'XML Stylesheet'

class clam.common.formats.**ZIPFormat**(*file*, ***kwargs*)

attributes = {}

mimetype = 'application/zip'

name = 'ZIP Archive'

CHAPTER 5

CLAM Parameters

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: –paramflag=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()

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()

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*)

validate(*values*)

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*)

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*)

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()

validate(*value*)

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

compilearg()

CHAPTER 6

CLAM Viewers

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*)

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

id = 'foliaviewer'

name = 'FoLiA Viewer'

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

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

id = 'tableviewer'

name = 'Table viewer'

read(*file*)

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

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

id = 'sonarviewer'

name = 'SoNaR Viewer'

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

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

id = 'xsltviewer'

name = 'XML Viewer'

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

CHAPTER 7

Installation

It's discouraged to download the zip packages or tarballs from github, install CLAM from the Python Package Index or use git properly.

Installation On Linux

Installation from the Python Package Index using the package manager *pip* it the recommended way to intall CLAM. 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 want to install globally, prepend the following commands with `sudo`:

CLAM can be installed from the Python Package Index using pip. Pip is usually part of the `python3-pip` package or similar. It downloads CLAM and all dependencies automatically::

```
$ pip3 install clam
```

If you already downloaded CLAM manually (from github), you can do:

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

If pip3 is not yet installed on your system, install it using: on debian-based linux systems (including Ubuntu):

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

on RPM-based linux systems:

```
$ yum install python3-pip
```

Note that sudo/root access is needed to install globally. Ask your system administrator to install it if you do not own the system. Alternatively, you can install it locally in a Python virtual environment:

```
$ virtualenv --python=python3 clamenv

$ . clamenv/bin/activate

(clamenv)$ pip3 install clam
```

It is also possible to use Python 2.7 instead of Python 3, adapt the commands as necessary.

CLAM also has some optional dependencies. For MySQL support, install `mysqlclient` using pip. For FoLiA support, install `FoLiA-Tools` using pip.

Installation on Mac OS X

Install a Python distribution such as Anaconda and follow the Linux instructions above.

Installation on Windows

CLAM does not support Windows, i.e. you can't run CLAM webservices on Windows. However, the CLAM Data API and client API will work, so clients connecting to CLAM webservices can run on Windows. Follow the same instructions as for Mac OS X.

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
```

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 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z