MetaDoc Client Documentation

Version 0.1.0

Bjørnar Grip Fjær

July 6, 2010

Contents

1 Introduction

MetaDoc is created as a way to securely transport data between a server and sites. It is created to improve the flow of information between High Performance Computing (HPC) sites and Uninett Sigma [1].

MetaDoc consists of a client, running at the site, and a server running at the Metacenter. MetaDoc takes care of authenticating the client on the server, packing and unpacking the data to and from Extensible Markup Language (XML), and transporting the data securely between client and server.

If you wish to get the client up and running as quickly as possible, the MetaDoc Client Quick Start Guide is a good place to start [2].

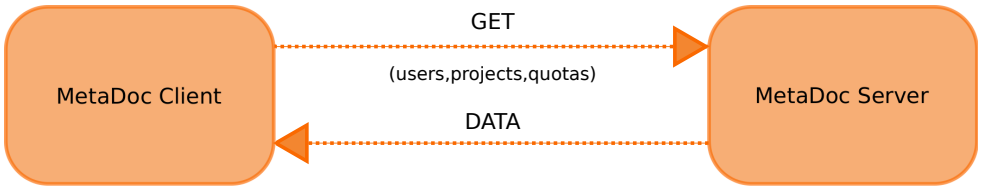

Figure 1: Client requesting data from server

Figure 1 shows how the client requests data from the server. The MetaDoc client provides the interface for retrieving this data, and it is then up to the site to decide how to handle the recieved data.

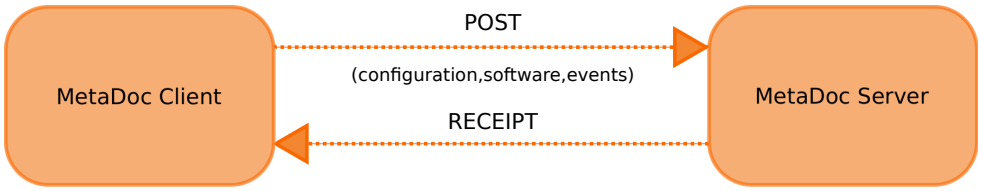

Figure 2: Client sending data to server. Server returns a receipt for recieved data.

In figure 2, data is sent from the client to the server. On recieving the data, the server will return a receipt, telling the client that the data has been processed and accepted or rejected.

1.1 Transmitting data

The MetaDoc API (MAPI) uses Secure Socket Layer (SSL)/Transport Layer Security (TLS) to transport data over Hypertext Transfer Protocol (HTTP) to ensure the data is safely encrypted. It uses X.509 certificates to identify the client to the server.

The data transmitted is packed to XML, which is a human readable data format[1]. It also has strong support in almost any commonly used programming language today.

1.2 Documentation overview

In section 2, an overview of the MetaDoc client is given. It explains what is done by the standard client, and how the client can run.

[1] At least for the technically inclined

Section 3 explains how to customize the MetaDoc client in order to process the data sent between the client and server.

Section 4 goes through the server side Application programming interface (API). It explains how the server acts in response to requests by the MetaDoc client.

The XML document used to send data between the client and server is explained in section 5.

Section 6 describes some useful classes and modules provided by the MetaDoc client for use when extending the client to send or recieve more data.

A guide to extending the client is given in section 7. This is a step for step guide that explains what is needed in order for the client to send or recieve more data.

The information flow between client and server is explained in more detail in section 8.

An explanation of possible errors that may occour during usage of the MetaDoc client is given in section 9.

2 Overview

Usage of the MetaDoc client is done mainly through the use of `mapi.py`, named after the MAPI. `mapi.py` takes care of sending and retrieving data to and from the server, caching any data that could not be sent, and validating XML data recieved.

2.1 Configuration

The MetaDoc client uses a configuration file `metadoc.conf`. This file *must* be placed in the same folder as the client itself. Listing 1 provides an example of a basic configuration file.

The configuration is in INI format and defines a section named `MetaDoc`.

Required parameters:

host baseurl for the MetaDoc Server that the client should communicate with.

key Absolute path to the clients X.509 certificate key file. This should be the private key for **cert**, and is used to encrypt data passed to the server.

cert Absolute path to the clients X.509 certificate file. This is used to authenticate the client with the server. See section 4.2 for more information.

site_name The name of the site the client is running on.

Optional parameters:

trailing_slash Whether the client should append a trailing slash at the end of URLs used to connect to the server. At the moment, this should be set to `True`.

valid Initially set to `False` when `mapi.py` creates a sample configuration file. This is set to avoid running the client without being properly configured. If this value is present, it *must* be set to `True` or `yes`.

`mapi.py` will create a initial configuration file if one is missing when it starts. This can be used as the basis for a configuration file.

Listing 1: Example of a basic configuration file

```
1  [MetaDoc]
2  host   = https://localhost:10000/metadoc/
3  key    = /home/bjornarg/metadoc/client/userkey.pem
4  cert   = /home/bjornarg/metadoc/client/usercert.pem
5  trailing_slash = True
6  valid = True
7  site_name = bjornarg
```

2.2 Handles

A handle is an option passed to the client when running the script.

`mapi.py` takes handles that tell the script what information to send or retrieve to or from the server. All handles can be mixed together *except* for handles that override each other. Handles that overrides others are explicitly stated below.

`mapi.py` takes the following handles:

-h, –help Displays a short help message explaning the handles that may be passed to `mapi.py`. Overrides any other handles.

-v, –verbose Verbose mode. Prints information about progress and information sent and recieved between client and server.

-q, –quiet Quiet mode. Prints nothing unless the program fails. Overrides **-v**, **–verbose**.

-l <log level>, –log-level=<log level> Sets the log level for the program. See section 2.3 for more information about what is logged at different levels.

-n, –no-cache Prevents the client from sending any cached data. If any errors occour when the client runs with this handle, data from this run will *not* be cached. For more information about caching, see section 2.4.

-e Sends event data from client to server.

-c Sends configuration data from client to server.

-s Sends software data from client to server.

-u Retrieves user data from the server.

-a Retrieves allocation data from server.

-p Retrieves project data from server.

–dry-run Does a dry run, not sending any data to server. Does not retrieve cached data, and does not save any data to cache. Should be run with verbose to see data that would be sent.

–all Sends and retrieves all possible data. Equal to -ecsuap.

–send-all Sends all possible data. Equal to -ecs.

–fetch-all Retrieves all possible data. Equal to -uap.

2.3 Logging

The client logs data to `/var/log/mapi/`. The folder must be readable and writable for the user that runs the. The client creates a new log file each day it runs, with the naming schema: `metadoc.client.YYYY-mm-dd.log`.

The client has five different logging levels. The list below gives an overview of what is logged at the different levels. The higher items in the list contain everything below as well, so that with a log level set to **error** will also contain **critical** logging.

The log level defaults **warning**.

debug Debugging information, used for development and error checking.

info Information about what is happening during execution, such as items sent or recieved to/from the server.

warning Warnings occouring during execution, mainly problems that will not cause a failure but that should be addressed.

error Errors that cause partial failure of the execution, such as being unable to connect to the server.

critical Critical failures that causes the execution to halt, or errors in the program code itself.

2.4 Caching

The client caches data to `/var/cache/mapi/`. Files are named after the data type that is cached in each file. The user running the client must have read and write access to this folder.

The client will cache any information that is not accepted by the server, *unless* the server returns a reciept for the information that marks the information as invalid or malformed in some way, such that the information will not be accepted if resent at a later date. See section 9 for more information about errors.

Data the client sends may be marked so that it will not resend any cached data when the client is run with the same handle. This is mainly for use for full updates, such as software and configuration, where any cached data would be outdated or duplicates if sent together with a new run.

If the **-n** or –**no-cache** handles are passed, the script will ignore any cached data completely and run as if it didn't exist. The client will also *not* cache any data on this run. The cached data will then be processed on the next run where **-n** or –**no-cache** is not passed.

3 MetaDoc Client API

The MetaDoc client calls a specified function based on the data passed between client and server. Only one type of data should be sent per document, and both the client and the server only checks for the expected type of data in the recieved XML document.

Each data type has a named container element within the XML document, which there should only be one of per document. If a list of data is passed between server and client, the list should be placed inside a container element. The name of the container element is used in naming modules and classes in order to ease readability of code.

An example of such a container is `users`, which holds `user_entry` elements. An example of a MetaDoc XML document with a list of users is shown in listing 2.

Listing 2: User list XML example

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <MetaDoc version="0.1.0" site_name="example">
3    <users>
4      <user_entry username="example1" />
5      <user_entry username="example2" uid="1" />
6    </users>
7  </MetaDoc>
```

The naming of the class that handles the data passed between server and client on the client side depends on whether data is passed from client to server, or the other way around.

3.1 Sending data

Before data can be sent, the XML document must be assembled. Because there is not always a standard way to populate this data on every site, a custom class is created. This class should be found under `custom.site<name>.Site<Name>`, where `<name>` is the name of the container element in the XML, and `<Name>` is the name capitalized (e.g. `config` would use `custom.siteconfig.SiteConfig`). This class should inherit from
`custom.abstract.MetaOutput`.

When the client is ready to fetch these items, the function `populate()` in this class is called. This function will gather the information to send, creating elements found in `<name>.definition.<Name>.legal_element_types` for this information and placing these items in `self.items`.

An example of such a class for the imaginary `posts` data is shown below.

Listing 3: custom.siteposts

```
1  from custom.abstract import MetaOutput
2  from posts.entries import PostEntry
3
4  class SitePosts(MetaOutput):
5      def populate(self):
6          """ Adds a couple of posts that should be sent. """
7          post_one = PostEntry(title="We're making progress!", author="bjornarg")
8          post_one.set_content("""This text will be in post_entry MetaElement
9  instance's self.text!""")
10         self.items.append(post_one)
11         post_two = PostEntry(title="Cleaning attributes" author="ProperAuthor")
12         post_two.set_content("We've made sure that author is sent in lower case.")
13         self.items.append(post_two)
```

The client will then take care of packing data to XML, sending the data to the server, processing the receipt returned from the server and caching any data that was not accepted by the server. For more information on caching, see section 2.4.

3.2 Recieving data

The client will take care of fetching the data from the server, unpacking the XML and creating objects based on the type of data recieved. Once this is done, a function called `process()` will be called on the class `custom.update<name>.Update<Name>`. When this function is called, the object's `self.items` should be a list of `<name>.definition.<Name>` objects.

Examples for producing files similar to the ones now in use based on information transferred through MetaDoc is given in `doc/examples/custom/`.

3.3 Summary

`process()` and `populate()` are the key functions that are used to customize how the client handles data passed between the MetaDoc client and server.

4 MetaDoc Server API

The MetaDoc server implements a REST-like API, however, there are certain differences from REST noted in section 4.3.

When the client performs a GET request on an availible URL, the server should return an XML document, or a HTTP status code refering to an error. The XML document should follow the MetaDoc Document Type Definition (DTD) [3]. Each URL only returns data from the requested data type. This means that a request to **baseurl/allocations/** will return an MetaDoc XML document containing only an `<allocations>` directly on the `<MetaDoc>` root, with `<all_entry>` tags as children of `<allocations>`. The client should disregard any information outside `<allocations>` when connecting to **baseurl/allocations/**.

In order to send data to the server, the client performs a POST request, with the POST data variable `metadoc` containing a MetaDoc XML document. The server will only accept data from the data type specified in the URL, and will disregard any other information. This means that a POST to **baseurl/events/** should be a MetaDoc XML document containing a `<events>` tag directly on the `<MetaDoc>` root, with any number of `<resourceUp>` and `<resourceDown>` tags as children of `<events>`.

When this data is sent to the server, the server should return a MetaDoc XML document containing a `<receipts>` tag, with a `<r_entry>` tag for each element recieved that has an **id**-attribute. This allows for a very fine grained error reporting.

4.1 Available URLs

/allocations/ Retrieves a list of allocations/quotas relevant to the client

/users/ Retrieves a list of users for the client

/projects/ Retrieves a list of projects relevant to the client

/config/ Sends system configuration to server

/events/ Sends site events to the server

/software/ Sends system software to server

4.2 Authentication

The server uses X.509 certificates to authenticate the client. In order for the site to be authenticated properly, the server must be aware of the client's certificate prior to the request, and the correct owner of the certificate must be saved on the server. This *must* be the same as the value for `site_name` set in the MetaDoc configuration (see section 2.1).

4.3 Differences from REST

There are certain differences in the API compared to the REST specification. The MetaDoc Server API makes use of HTTP POST where HTTP PUT should be used in accordance with REST. This is due to limitations in standard Python libraries.

Because the MetaDoc Server does not give the client access to delete or replace data on the server, this does not create problems for the Server API.

4.4 Server HTTP responses

The server makes use of HTTP status codes to identify what error has occoured if the server is unable or unwilling to process the request from the server. Table 1 contains a list of status codes the server returns, and why these status codes occour.

Table 1: List of HTTP status codes used by the MetaDoc Server

Code	**Official name**	**MAPI-Reason**
200	200 OK	The server has processed the request and should return an XML document.
400	400 Bad Data	The data excepted from the client was not sent, or the XML document did not validate against the DTD or was malformed XML.
403	403 Forbidden	The client certificate was not recognized as an authorative source for the site given in the XML document.
404	404 Not found	The URL the client is attempting to access is not available on the server. Check host in `metadoc.conf`. If host is correct, make sure client and server uses same version. See section 7.4 for more information on version differences.
500	500 Server Error	The server failed to process the request. This does not include errors that return a receipt.
501	501 Not Implemented	The request method is not implemented for this URL. This will happen if the client attempts to use a request method that is not implemented for this URL, such as using the POST request method against an URL that only supports GET.

5 XML document

The XML document should follow the form described in the MetaDoc DTD [3]. Below certain conventions used in the XML build is explained. Any alterations to the DTD should follow these conventions in order for the client and server to continue functioning normally.

5.1 Document build

Any type of information sent should only create one direct child of the root element, `<MetaDoc>`. This means that when lists of information is sent, the list elements should be placed within a container element, and *not* directly in the root element. The container element should have an self explanatory name about the information passed.

An example is that `<user_entry>` elements are placed within a `<users>` element. Here `<users>` is considered the container element, and there should only be one of them in each MetaDoc XML document. `<users>` may contain any number of `<user_entry>` elements.

5.2 Dates

All dates in the document should be on the form specified by RFC3339 [4]. The `utils` module provides a function `date_to_rfc3339` that takes a `datetime.datetime` object and returns a string on RFC3339 form. It also provides a function `rfc3330_to_date` which will return a `datetime.datetime` object from a proper RFC3339 string, or **False** if the string is not a correct RFC3339 date.

5.3 Special attributes

The **id** attribute of elements have a special function in MetaDoc. This attribute is used to identify the object when recieving receipts from the server whether elements have been added. The attribute is *not* saved in caching to avoid duplicate **id**s when resending cached data together with new data. If you want to give elements a special identifier that should be saved, it must be called something other than **id**.

6 Useful classes and modules

The MetaDoc client defines a series of classes and modules that are used to define the XML data passed between client and server.

`MetaElement` refers to `metaelement.MetaElement`, and `MetaDoc` refers to `metadoc.MetaDoc`.

6.1 MetaDoc

The class `MetaDoc` is used to define the document itself. It provides functionality to alter the document structure, find elements within the document and generate XML data from the document. It contains a series of `MetaElement` sub classes that defines the content within the document.

6.2 MetaElement

`MetaElement` is used to define content within the document. The class is ment to be a parent class for other classes that defines particular elements. An element is equal to an XML tag, such as `<users>` or `<resourceUp>` in a MetaDoc XML document. An instance of such a sub class is used to define a particular tag in the XML document, with attributes and content.

6.2.1 Class variables

xml_tag_name Each `MetaElement` sub class should define a class variable **xml_tag_name**, that should be a string containing the name of the XML tag the class describes. For the sub class defining the `<users>` tag, this should be set to "users".

url A `MetaElement` sub class that defines a main container element, that is, an element that is placed as a direct child of the root node `<MetaDoc>` in the XML document, should also define a class variable **url** that is a string containing the particular part of the URL that is used to send or retrieve information for the data type passed. If the type of data is a list of users, and it should retrieve the list of users from the url **/users/**, the **url** class variable should be set to "users".

The classes that define a main container element should also define either the class variable **update_handler** or **site_handler**, depending on whether the data is ment to be recieved from the server or sent from the client, respectively.

update_handler **update_handler** should be a sub class of `custom.abstract.MetaInput`. This class should be placed in `custom.update<name>.Update<Name>`, so for users this would be `custom.updateusers.UpdateUsers`. When data of the type defined by the `MetaElement` sub class is recieved, an instance of **update_handler** will be created, and the instance's **self.items** will be populated with a list of `MetaElement` sub classes. Then the **update_handler**'s `process()` function will be called. Normally, **self.items** should be of length 1.

site_handler **site_handler** should be a sub class of `custom.abstract.MetaOutput`. This class should be placed in `custom.site<name>.Site<Name>`, so for events this would be `custom.siteevents.SiteEvents`. When the script is called to send data of the type defined by the `MetaElement` sub class, an instance of **site_handler**

is created, and the instance's `populate()` function is called. `populate()` should populate the instance's **self.items** with a list of `MetaElement` sub classes. When `populate()` is done, **self.items** is added to the `MetaElement` sub class instance's **self.sub_elements**.

6.2.2 Allowed sub_elements

There are certain restrictions on what classes can be placed within a `MetaElement` sub class instance's **self.sub_elements**, because not every XML tag can have any other XML tag as children. A `MetaElement` sub class therefor defines a **self.legal_element_types**. This should be a list of `MetaElement` sub classes that are allowed to be children of the XML current `MetaElement` sub class.

As an example, if `Users` is a `MetaElement` sub class defining the `<users>` XML tag, and `UserEntry` is a `MetaElement` sub class defining the `<user_entry>` XML tag, which can be a child of `<users>` in the XML document, a `Users` instance would have `UserEntry` in it's **self.legal_element_types**.

6.2.3 Tag attributes

A tag may have attributes set. These should be defined in the `MetaElement` sub class' `__init__()` function. They *must* have the same name as the attribute has in the XML document. If an attribute is optional in the XML element, it should also be optional in `__init__()`.

The sub class should in it's `__init__()` function figure out which attributes are available and which are not, and pass these on to the `MetaElement` `__init__()` function through `super()`.

6.2.4 Cleaning functions

The `MetaElement` class defines a couple of useful cleaning functions that are commonly used in cleaning attribute values. These functions may be called inside an attribute's clean function. The errors raised should in most cases not be caught inside the clean function, unless you are able to fix the attribute in some way if these functions cannot. The errors will be caught by the client and the element rejected.

_clean_date() Takes a `date`, `datetime`, `time.time()` or string. If the argument is any of the three first types, it will convert it to a RFC3339 date. If it is a string, it will check that the date is a proper RFC3339 date.

This function will raise an `metaelement.IllegalAttributeValueError` if an illegal type or non-RFC3339 string is passed.

_clean_allowed_values() Takes the value and a list of allowed values and checks whether the value is in the list. Can perform case insensitive matching.

Will raise an `metaelement.IllegalAttributeValueError` if the value is not in the list.

6.3 UniqueID

The `utils` module provides a class called `UniqueID` that can provide a unique identifier to objects passed through the function `get_id()`. This should be used for entries passed from client to server to set as the **id** attribute so that the server can properly identify the entry when returning a receipt.

`get_id()` returns a string with an increasing number prefixed by an underscore (_). The underscore is present because XML does not allow a number as an **id** attribute.

6.4 Examples

6.4.1 Connection figure

Figure 3 shows an example of how these connections work. Here the definition of projects is shown, with connections to the XML document, DTD, server URL and **update_handler**.

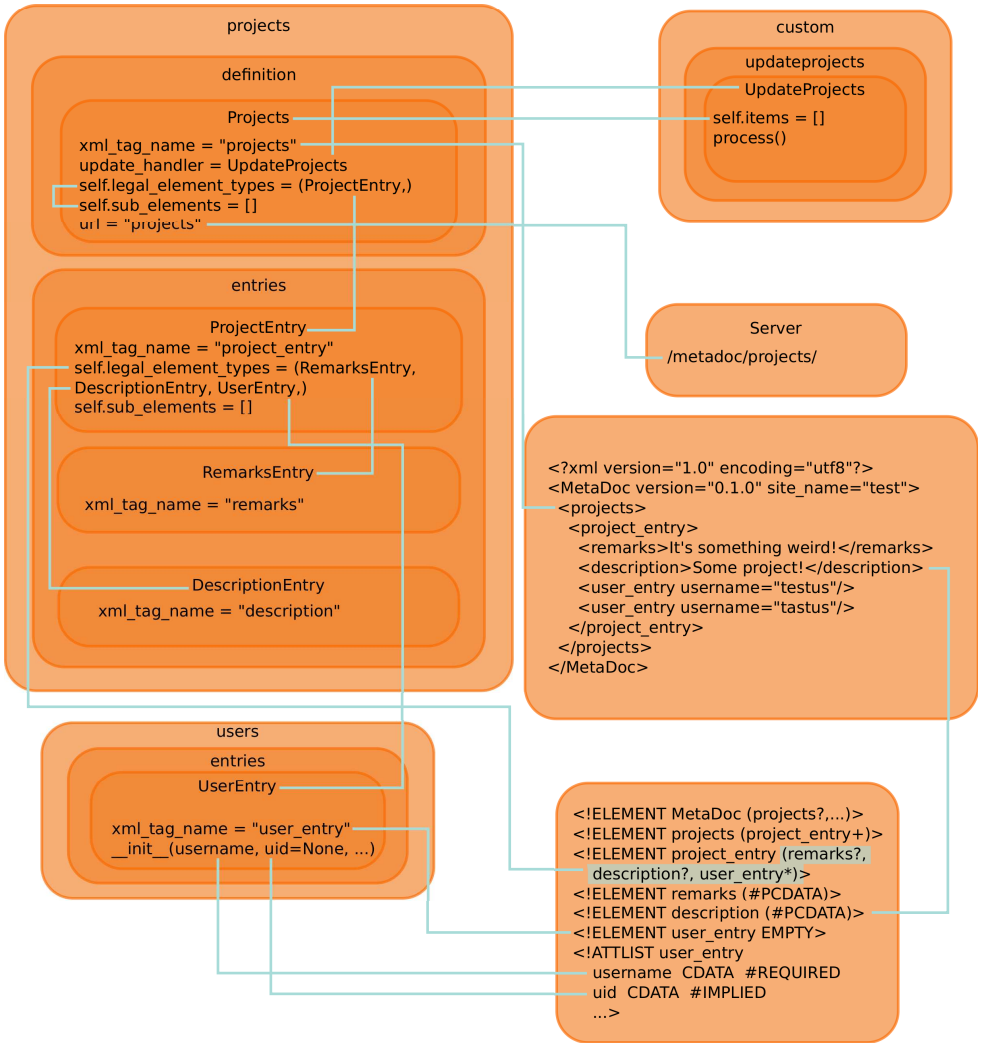

Figure 3: Example of how projects data is defined and connection between classes used in definition and processing of project data.

6.4.2 Script example

Below a small example of transferring posts from client to server is shown.

Listing 4: posts.definition

```
import metaelement
from custom.siteposts import SitePosts
from posts.entries import PostEntry

class Posts(metaelement.MetaElement):
    xml_tag_name = "posts"
    site_handler = SitePosts
    url = "posts"

    def __init__(self):
        super(Posts, self).__init__(Posts.xml_tag_name)
        self.legal_element_types = (PostEntry,)
```

Listing 5: posts.entries

```
import metaelement
import re
from custom.siteposts import SitePosts
from utils import UniqueID()

class PostEntry(metaelement.MetaElement):
    xml_tag_name = "post_entry"

    def __init__(self, title, author):
        unique_id = UniqueID()
        attributes = {
            'title': title,
            'author': author,
            'id': unique_id.get_id(),
        }
        super(PostEntry, self).__init__(PostEntry.xml_tag_name, attributes)
    def set_content(self, content):
        """ Sets the content of the post. """
        self.text = content
    def clean_author(self, author):
        """ Makes sure the author's name is in lower case. """
        return author.lower()
```

Listing 6: custom.siteposts

```
from custom.abstract import MetaOutput
from posts.entries import PostEntry

class SitePosts(MetaOutput):
    def populate(self):
        """ Adds a couple of posts that should be sent. """
        post_one = PostEntry(title="We're making progress!", author="bjornarg")
        post_one.set_content("""This text will be in post_entry MetaElement
instance's self.text!""")
        self.items.append(post_one)
        post_two = PostEntry(title="Cleaning attributes" author="ProperAuthor")
        post_two.set_content("We've made sure that author is sent in lower case.")
        self.items.append(post_two)
```

Listing 7: XML Example

```
<?xml version="1.0" encoding="utf-8"?>
<MetaDoc version="0.0.0" site_name="example">
  <posts>
    <post_entry title="We're making progress!" author="bjornarg" id="_1">
      This text will be in post_entry MetaElement instance's self.text!
    </post_entry>
    <post_entry title="Cleaning attributes" author="properauthor" id="_2">
      We've made sure that author is sent in lower case.
    </post_entry>
  </posts>
</MetaDoc>
```

7 Extending MetaDoc

MetaDoc is explicitly designed to allow for future enhancements in case more information should be sent. In order to do so, an series of steps are required. The list below details each of these steps, and each step is explained in more detail in the following sections.

1. Extend the MetaDoc DTD with the definition of the data.
2. A definition file explaining the data on the client.
3. An entries file, explaining any entries allowed in the data on the client.
4. A `MetaInput` or `MetaOutput` sub class that should handle data, depending on whether data is recieved or sent to or from the server, respectively.
5. Adding a handle to `mapi.py` that will activate the data type.
6. Configuring the server to send or recieve the intended data.

Figure 3 on page 13 shows an example of how the data for projects is defined, and the connection between classes that are used when data about projects is recieved from the server.

7.1 Altering DTD

The XML document follows certain conventions that should be followed when extending the DTD. These conventions are explained in more detail in section 5.

Before you alter the DTD you should know exactly what data should be sent. Create an `<!ELEMENT>` with a descriptive name of the data sent. As an example, `<users>` is used for a list of users.

Add any attributes necessary to describe the set of data. If the data is a list of entries, such as a list of users, create an `<!ELEMENT>` as a possible sub-element that contains the information about each entry. Any short information about the entry should be placed in attributes of the entry. If there is more information, such as information that could be several sentences or lines, it should not be placed as an attribute. This information should be placed inside the element itself. If there are several types of long information for each entry, create a descriptive `<!ELEMENT>` for each as a sub-element of each entry to contain the text. Otherwise the text may be placed directly inside the entry element itself.

As an example, the MetaDoc DTD [3] defines `project_entry`, where `remarks` and `description` are allowed sub-elements that contain any text. Meanwhile, `resourceUp` and `resourceDown` places the text directly inside the tag itself, as not other information is allowed inside these tags.

7.2 Defining the data client side

Add a package to the client with the name of the main element [5]. Create a module `definition` inside this package. `definition` should define the main element with a sub class of `metaelement.MetaElement`.

Create a module `entries` inside the same package. This file should contain definitions of each entry, and potential sub elements for each entry, as a sub class of `metaelement.MetaElement`.

Add the entry class(es) to `self.legal_element_types` of the `metaelement.MetaElement` sub class defining the main element.

`metaelement.MetaElement` sub classes may define a `clean_<attribute name>()` for each attribute on the element. This method will recieve the attribute value, and should return the attribute value after any potential cleaning is done on it. Please note that *all* attribute values *must* be strings, so if any value set as an attribute might be set as anything other than a string, the clean-function is the place to convert it. If the attribute contains a value that is not allowed, a sub class of `metaelement.IllegalAttributeError` should be raised that defines the error that has occoured.

The `metaelement.MetaElement` defines some methods that are commonly used in cleaning methods, such as converting a `date` object into an RFC3339 string, or checking for legal values of an attribute. See section 6 for more information about these functions and how to build these classes.

7.3 Custom client handles

If the data is to be sent from client to server, create a module `custom/site<main element name>.py` that contains a sub class of `custom.abstract.MetaOutput`. This class should define a method `populate()` which gathers the information to be sent from the site and appends an instance of a entry-class, as defined in section 7.2, to **self.items** for each entry.

If the data is sent from server to client, a module called `custom/update<main element name>.py` should be created that contains a sub class of `custom.abstract.MetaInput`. This class should define a method `process()` that processes any recieved data in **self.items**.

See section 3 and 6 for more information on these classes.

7.4 Versioning

MetaDoc passes a **version** attribute on it's root element, `<MetaDoc>` when sending information between client and server. This version is a string on the form "`X.Y.Z`", where `X`, `Y` and `Z` are numbers. Changes made to each number indicate different levels of breakage.

When `X` is changed, changes are made such that the current information passed is changed in some way. This may be changes to the DTD where any of the currently passed information is affected (addition/removal of attributes, changes to how attribute values are presented or should be parsed, addition/removal of sub-elements). If the client or server encounters a document with a different value of `X` in the version number, it should *not* accept the data, as it cannot be sure it will handle it correctly.

Changes to `Y` indicates a change that does *not* change the current behaviour in any way, but may be instances where new information might be passed. When the client or server encounters a document with a different value of `Y` it should log a warning, but otherwise proceed as normal.

`Z` is currently not used for anything, but is present for potential usability in the future. Differences in `Z` should be logged as debug information.

8 Information flow

The client will always be the initiator in either requesting data from the server or sending data.

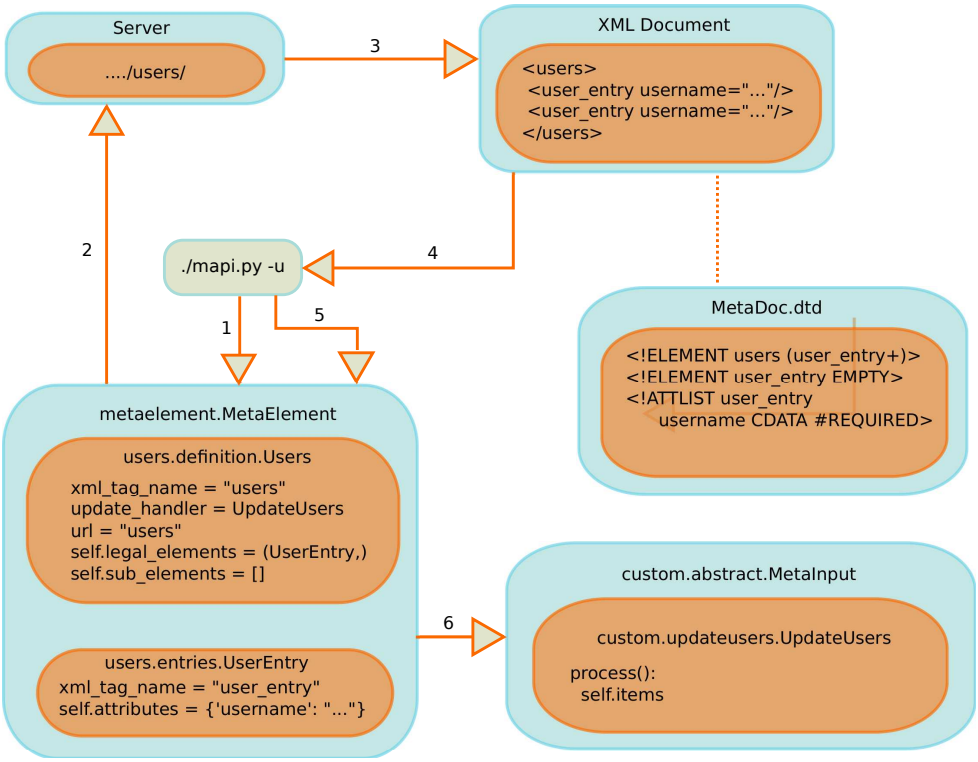

Figure 4: Information flow when requesting user list with MetaDoc

Figure 4 shows how information passes when a client requests a user list. The steps are as follows:

1. `mapi.py` is run with the handle **`-u`**, which will check `users.definition.Users` for which URL to access on the server to retrieve the information.
2. A request is sent to the server to retrieve the information.
3. The server creates an XML document with a list of users for the site.

   The XML document sent is defined by the MetaDoc DTD.
4. `mapi.py` recieves the XML document, checks that it is valid according to the DTD.
5. If the XML document passes validation, an instance of `users.definition.Users` is created, and an instance of `users.entries.UserEntry` is created for each `<user_entry>` in the XML document.

   `users.definition.Users` and `users.entries.UserEntry` may validate attributes and refuse to create any elements where attribute validation does not pass.

6. The list of validated `users.entries.UserEntry` instances is placed within `self.items` for an `custom.updateusers.UpdateUsers` instance, and the `process()` function is called for the processing of the user list.

8.1 Validation

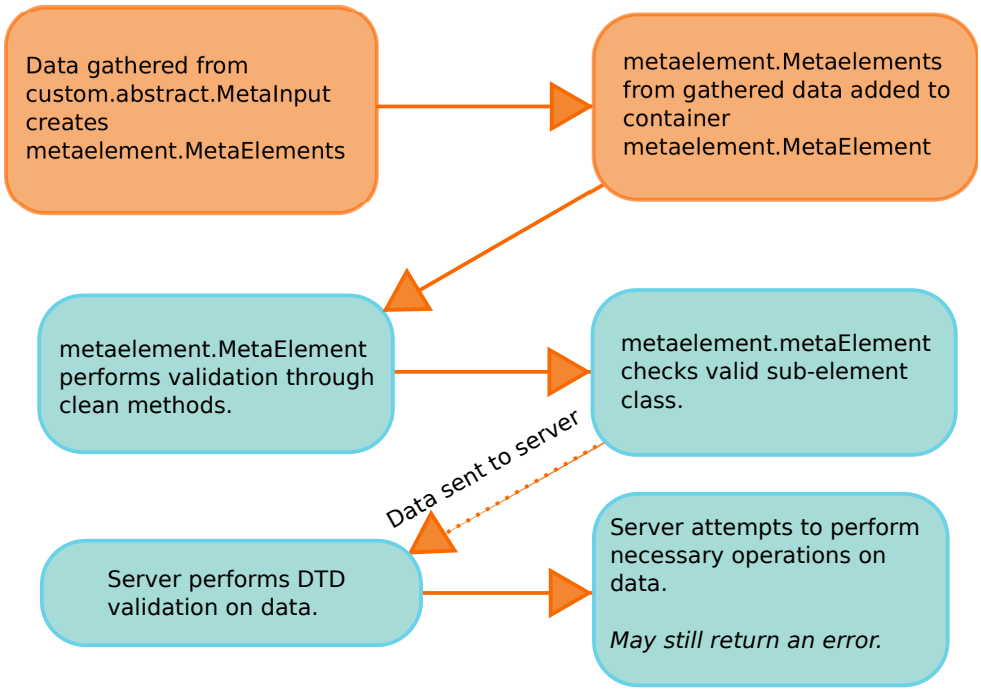

Figure 5: Shows validation procedures when data is passed from client to server

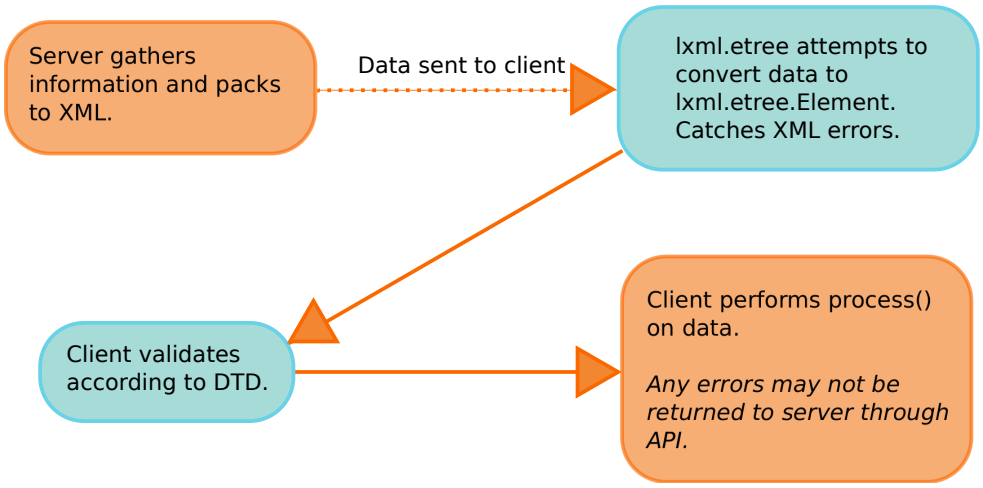

Figure 6: Shows validation procedures when data is passed from server to client

9 Errors

The server returns a `<receipt>` containing an `<r_entry>` *for each* element passed. The `<r_entry>` has the required attributes **id** and **code**, containing the ID of the element sent by the client and the error code, respectively. It may also contain an attribute **note** with a short note explaining the error if extra information is availible. The `<r_entry>` tag might also contain text with a longer message, if more information is needed about the error.

An example of a error message would be an error code `2001` with the note `Missing attribute "reason"`.

9.1 Document errors

In the special case where there are problems with the document itself, such as XML errors or the document not passing DTD verification, the `<r_entry>` **id** attribute will be set to `0` (zero), reffering to the document itself.

References

[1] Austad, Henrik, *Improving the Information Flow within the Metacenter*, `http://www.austad.us/metadoc/improvingFlow.pdf`

[2] Fjær, Bjørnar Grip, *MetaDoc Quick Start Guide*, `http://bjornar.me/metadoc/quick_start_guide.pdf`

[3] *MetaDoc Document Type Definition*, `http://bjornar.me/metadoc/MetaDoc.dtd`

[4] *RFC3339*, `http://www.ietf.org/rfc/rfc3339.txt`

[5] *Python modules*, `http://docs.python.org/tutorial/modules.html`

List of Figures

List of Tables

Listings

Glossary

API Application programming interface. 2, 8

DTD Document Type Definition. 8–10, 13, 15–17, 19

HPC High Performance Computing. 1

HTTP Hypertext Transfer Protocol. 1

MAPI MetaDoc API. 1, 3

SSL Secure Socket Layer. 1

TLS Transport Layer Security. 1

XML Extensible Markup Language. 1–3, 6–13, 15, 17, 19, 22

A List server return codes

Table 2: Return codes recieved from server

Return code code	Meaning	Extra notes
1000	No error	
2000	Error with the XML data	
2001	Missing attribute	Missing attribute should be returned as a note.
5000	Database error	
5001	MySQL database error	Note should contain the MySQL error code, and the message the MySQL error message
6001	Another element error	Another element in the same set has been rejected, and this type of set will not accept any elements if any element contains an error

B Included examples

`doc/examples/` includes a set of examples for using MetaDoc. Below is a list of the included examples and what they do.

cli/event.py A command line interface for adding events. Should be placed inside `client/` when run. See `event.py --help` for usage.

custom/updateusers.py Custom function for converting recieved user data into a passwd/shadow file.

custom/updateprojects.py Custom function for converting recieved project data to a project user file.

custom/updateallocations.py Custom function for converting recieved allocation data into a quota file for projects.