MetaDoc Documentation

Bjørnar Grip Fjær

June 25, 2010

Contents

1 Using the MetaDoc client

Usage of the MetaDoc client is done mainly through the use of `main.py`. `main.py` takes care of sending and retrieving data from the server, as well as caching any data that could not be sent.

When `main.py` should send data to the server, a custom function that should populate the data to be sent is called, so that each site can customize the way data is gathered on the site.

When `main.py` recieves data from the server, it calls a custom function based on the data recieved, where each site can define what should be done with the recieved data.

1.1 Handles

`main.py` takes handles that tells 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.

`main.py` takes the following handles:

-h, –help Displays a short help message explaning the handles that may be passed to `main.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 1.3 for more information about what is logged at different levels.

-n, –no-cache Prevents the client from sending any cached data. For more information about caching, see section 1.2.

-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.

1.2 Caching

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 6 for more information.

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 This is mainly for use for full updates, such as software and configuration, where any cached data would be outdated or duplicates if sent together with a new run.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 cached data will then be processed on the next run where **-n** or –**no-cache** is not passed.

1.3 Log levels

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.

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.

The log level defaults to the lowest possible, so everything will be logged if nothing is set.

1.4 Customizing MetaDoc

1.4.1 Sending data

To send data to the server, the client creates instances of a sub-class of the `custom.abstract.MetaOutput` class. These classes should define a `populate()` function that populates `self.items` with a set of `metaelement.MetaElement` sub-class instances. Once `self.items` is populated, the server packs it to XML and sends it to the server.

The server *must* return a reciept for each entry, specifying whether the entry has been accepted by the server, or return an error code, as defined i table 1, if the entry is not accepted. See section 6 and 1.2 for more information.

1.4.2 Recieving data

When the client recieves data from the server, it parses the data and places the parsed data in a `custom.abstract.MetaInput` instance. Which `MetaInput` sub-class is used is defined in the element's description by the class variable `update_handler`.

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

2 Extending MetaDoc

To extend MetaDoc to send more data, the following is needed. Each step is explained in more detail below. Certain restrictions is set on how the XML document should be formed. See section 3 for more information.

- Definition of data to be sent in the MetaDoc DTD.
- A definition file explaining the data on the client.
- An entries file, explaining any entries allowed in the data on the client.
- A `MetaInput` or `MetaOutput` sub-class that should handle data, depending on whether data is recieved or sent to or from the server, respectively.
- Adding a handle to `main.py` that will activate the data type.
- Configuring the server to send or recieve the intended data.

2.1 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 3.

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.

2.2 Defining the data on the client

Add a module to the client with the name of the main element. Create a file `definition.py` inside this module. `definition.py` should define the main element with a subclass of `metaelement.MetaElement`.

Create a file `entries.py` inside the same module. 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.

2.3 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 2.2, to `self.items` for each entry.

2.4 Versioning

MetaDoc passes a **version** attribute on it's root element, `<MetaDoc>` when sending information between client and server. This version is a number 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 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 normally.

`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.

3 XML document

The XML document should follow the form described in the MetaDoc DTD [1].

3.1 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.

An example is that `<user_entry>` elements are placed within a `<users>` element.

3.2 Element attributes

All element attributes *must* be strings. This is because the attributes must be placed in the XML document, and without knowing the way to represent the attribute as a string it is not possible to properly use it as one.

After the attributes `clean`-function is run, it will be checked that the attribute value is a **basestring** (**unicode** or **str**). If any attribute is not, the element will not be sent.

3.3 Dates

All dates in the document should be on the form specified by RFC3339 [2]. 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.

3.4 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**.

3.5 Client

3.5.1 Cleaning attributes

When an element is added as a sub-element to `metaelement.MetaElement`, a clean function is called for each attribute. The element defenition for sub-elements added may implement a function called `clean_<attribute name>`. This function should validate that the value of the attribute and make sure it returns the string value of the attribute. This makes it possible to create elements by passing non-string variables, such as `datetime` objects for date fields, then converting them in the clean function.

4 MetaDoc API description

4.1 Server API

The MetaDoc server implements a REST-like API. The server defines several URLs that can be accessed from the client:

baseurl/allocations/ Retrieves a list of allocations relevant to the site

baseurl/users/ Retrieves a list of users for the site

baseurl/projects/ Retrieves a list of projects relevant to the site

baseurl/config/ Sends system configuration to server

baseurl/events/ Sends site events to the server

baseurl/software/ Sends system software to server

When sending information to the MetaDoc Server, only the information relevant to that URL is processed. Any XML data sent that is not relevant for that URL is discarded, e.g. event information sent to **/baseurl/config/** will be discarded by the server. No receipt will be returned for this data.

The server will return a MetaDoc XML document containing a `<reciept>` element, which will contain `<r_entry>` elements for each element recieved. The `<r_entry>` element should return a code from table 1 for each element. See section 6 for more information on errors.

4.1.1 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 access the MetaDoc Server API gives to the client is limited, this change does not prohibit any other functionality.

4.1.2 Server HTTP responses

The server makes use of HTTP status codes.

If the client does not send a SSL certificate, sends a sertificate unknown to the server, or attempts to get information about sites not identified with the certificate, the server returns a “403 Forbidden“ status code.

5 Information flow

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

Figure 1 shows how information passes when a client requests a user list. The steps are as follows:

1. `main.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. `main.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.

6 Errors

The server returns a `<receipt>` containing an `<r_entry>` for each element parsed. The `<r_entry>` has the required attributes **id** and **code**, containing the ID of the element 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.

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

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

A List of errors

Table 1: Error codes recieved from server

Error code	Meaning	Extra notes
1000	No errors	
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

B Information flow

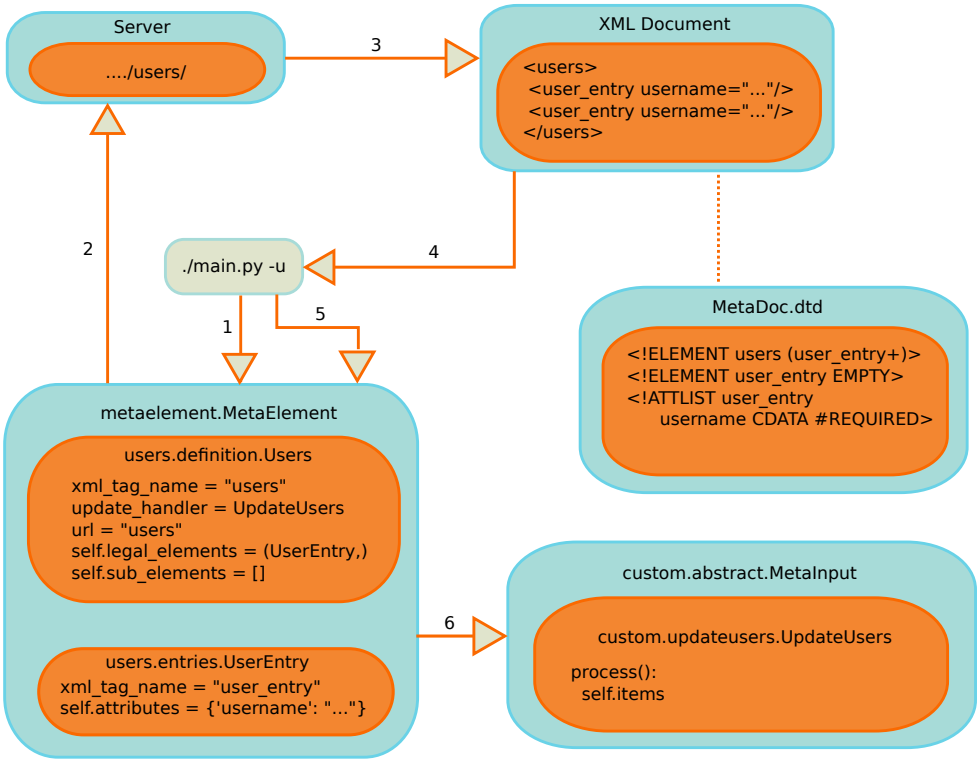

Figure 1: Information flow when requesting user list with MetaDoc