katportalclient Documentation

Release 0.2.3.dev412+head.ab525bf

SKA SA - CAM

Mar 01, 2022

Contents

A client for simple access to **katportal**, via websocket and HTTP connections. The HTTP methods allow once-off requests, like the current list of schedule blocks. For continuous updates, use the Pub/Sub methods, which work over a websocket.

CHAPTER 1

Dependencies

Details can be found in *setup.py* but basically it is only:

- katversion
- tornado is used as the web framework and for its asynchronous functionality.

Note: `setup.py` depends on `katversion`, so make sure that is installed before installing the package.

CHAPTER 2

Install

```
pip install katportalclient
```

CHAPTER 3

Example usage

See the *examples* folder for code that demonstrates some usage scenarios.

CHAPTER 4

Contents

4.1 katportalclient

4.1.1 client

Websocket client and HTTP module for access to katportal webservers.

exception katportalclient.client.**InvalidResponseError**
Bases: exceptions.Exception

Raise if server response was invalid.

class katportalclient.client.**KATPortalClient**(*url*, *on_update_callback*, *io_loop=None*, *logger=None*)
Bases: future.types.newobject.newobject

Client providing simple access to katportal.

Wraps functions available on katportal webservers via the Pub/Sub capability, and HTTP requests.

Parameters

url: str

Client sitemap URL: http://<portal server>/api/client/<subarray #>.
E.g. for subarray 2: http://1.2.3.4/api/client/2
(**Deprecated**: use a websocket URL, e.g. ws://...)

on_update_callback: function Callback that should be invoked every time a Pub/Sub update message is received. Signature has to include a single argument for the message, e.g. *def on_update(message)*.

io_loop: tornado.ioloop.IOLoop Optional IOLoop instance (default=None).

logger: logging.Logger Optional logger instance (default=None).

Attributes

is_connected Return True if websocket is connected.

sitemap Returns the sitemap using the URL specified during instantiation.

sub_nr Returns subarray number, if available.

Methods

add(*args, **kwargs)	Simple method useful for testing.
authorized_fetch(*args, **kwargs)	Wraps tornado.fetch to add the Authorization headers with the locally cached session_id.
connect(*args, **kwargs)	Connect to the websocket server specified during instantiation.
create_userlog(*args, **kwargs)	Create a userlog with specified linked tags and content, start_time and end_time.
disconnect(self)	Disconnect from the connected websocket server.
future_targets(*args, **kwargs)	Return a list of future targets as determined by the dry run of the schedule block.
get_sitemap(*args, **kwargs)	Returns the sitemap using the URL specified during instantiation.
get_sub_nr(*args, **kwargs)	Returns subarray number, if available.
login(*args, **kwargs)	Logs the specified user into katportal and caches the session_id created by katportal in this instance of KatportalClient.
logout(*args, **kwargs)	Logs user out of katportal.
modify_userlog(*args, **kwargs)	Modify an existing userlog using the dictionary provided as the modified attributes of the userlog.
sb_ids_by_capture_block(*args, **kwargs)	Return list of observation schedule blocks associated with the given capture block ID.
schedule_block_detail(*args, **kwargs)	Return detailed information about an observation schedule block.
schedule_blocks_assigned(*args, **kwargs)	Return list of assigned observation schedule blocks.
sensor_detail(*args, **kwargs)	Return detailed attribute information for a sensor.
sensor_history(*args, **kwargs)	Return time history of sample measurements for a sensor.
sensor_names(*args, **kwargs)	Return list of matching sensor names.
sensor_subarray_lookup(*args, **kwargs)	Return full sensor name for generic component and sensor names.
sensor_value(*args, **kwargs)	Return the latest reading of a sensor.
sensor_values(*args, **kwargs)	Return a list of latest readings of the sensors matching the specified pattern.
sensors_histories(*args, **kwargs)	Return time histories of sample measurements for multiple sensors.
set_sampling_strategies(*args, **kwargs)	Set up a specified sensor strategy for a filtered list of sensors.
set_sampling_strategy(*args, **kwargs)	Set up a specified sensor strategy for a specific single sensor.
subscribe(*args, **kwargs)	Subscribe to the specified string identifiers in a namespace.
unsubscribe(*args, **kwargs)	Unsubscribe from the specified string identifiers in a namespace.

Continued on next page

Table 1 – continued from previous page

userlog_tags(*args, **kwargs)	Return all userlog tags in the database.
userlogs(*args, **kwargs)	Return a list of userlogs in the database that has an start_time and end_time combination that intersects with the given start_time and end_time.

add(**args*, ***kwargs*)
Simple method useful for testing.

authorized_fetch(**args*, ***kwargs*)
Wraps tornado.fetch to add the Authorization headers with the locally cached session_id.

connect(**args*, ***kwargs*)
Connect to the websocket server specified during instantiation.

create_userlog(**args*, ***kwargs*)
Create a userlog with specified linked tags and content, start_time and end_time.

Parameters

content: str The content of the userlog, could be any text. Required.

tag_ids: list A list of tag id's to link to this userlog. Example: [1, 2, 3, ..] Default: None

start_time: str A formatted datetime string used as the start time of the userlog in UTC. Format: %Y-%m-%d %H:%M:%S. Default: None

end_time: str A formatted datetime string used as the end time of the userlog in UTC. Format: %Y-%m-%d %H:%M:%S. Default: None

Returns

userlog: dict The userlog that was created. Example: {

'other_metadata': [], 'user_id': 1, 'attachments': [], 'tags': '[]', 'timestamp': '2017-02-07 08:47:22', 'start_time': '2017-02-07 00:00:00', 'modified': '', 'content': 'katportalclient userlog creation content!', 'parent_id': '', 'user': {'email': 'cam@ska.ac.za', 'id': 1, 'name': 'CAM'}, 'attachment_count': 0, 'id': 40, 'end_time': '2017-02-07 23:59:59'

}

disconnect(*self*)
Disconnect from the connected websocket server.

future_targets(**args*, ***kwargs*)
Return a list of future targets as determined by the dry run of the schedule block.

The schedule block will only have future targets (in the targets attribute) if the schedule block has been through a dry run and has the verification_state of VERIFIED. The future targets are only applicable to schedule blocks of the OBSERVATION type.

Parameters

id_code: str Schedule block identifier. For example: `20160908-0010`.

Returns

list: Ordered list of future targets that was determined by the verification dry run. Example: [

{ "track_start_offset":39.8941187859, "target":"PKS 0023-26 | J0025-2602 | OB-238, radec, "

"0:25:49.16, -26:02:12.6, " "(1410.0 8400.0 -1.694 2.107 -0.4043)",

"track_duration":20.0

}, {

"track_start_offset":72.5947952271, "target":"PKS 0043-42 | J0046-4207, radec, "

"0:46:17.75, -42:07:51.5, " "(400.0 2000.0 3.12 -0.7)",

"track_duration":20.0

}, {

"track_start_offset":114.597304821, "target":"PKS 0408-65 | J0408-6545, radec, "

"4:08:20.38, -65:45:09.1, " "(1410.0 8400.0 -3.708 3.807 -0.7202)",

"track_duration":20.0

}

]

Raises

ScheduleBlockTargetsParsingError: If there is an error parsing the schedule block's targets string.

ScheduleBlockNotFoundError: If no information was available for the requested schedule block.

get_sitemap(**args*, ***kwargs*)

Returns the sitemap using the URL specified during instantiation.

The portal webserver provides a sitemap with a number of URLs. The endpoints could change over time, but the keys to access them will not. The websever is only queried once, the first time the property is accessed. Typically users will not need to access the sitemap directly - the class's methods make use of it.

The sitemap can also be accessed synchronously via the `sitemap()` property, but that may block the Tornado event loop the first time it used.

Returns

dict: Sitemap endpoints, will include at least the following:

```
{ 'websocket': str,
  'historic_sensor_values': str,
  'schedule_blocks': str,
  'capture_blocks': str,
  'sub_nr': str,
  ... }

websocket: str
    Websocket URL for Pub/Sub access.
historic_sensor_values: str
    URL for requesting sensor value history.
schedule_blocks: str
    URL for requesting observation schedule block␣
↪information.
```

(continues on next page)

(continued from previous page)

```
capture_blocks: str
    URL for requesting observation capture block␣
↪information.
sub_nr: str
    Subarray number to access (e.g. '1', '2', '3', or '4').
subarray_sensor_values: str
    URL for requesting once off current sensor values.
target_descriptions: str
    URL for requesting target pointing descriptions for a
    specified schedule block
```

get_sub_nr(**args*, ***kwargs*)

Returns subarray number, if available.

This number is based on the URL used to connect to katportal, provided during instantiation.

Returns

sub_nr: int Subarray number.

Raises

SubarrayNumberUnknown:

- If the subarray number could not be determined.

is_connected

Return True if websocket is connected.

login(**args*, ***kwargs*)

Logs the specified user into katportal and caches the session_id created by katportal in this instance of KatportalClient.

Parameters

username: str The registered username that exists on katportal. This is an email address, like abc@ska.ac.za.

password: str The password for the specified username as saved in the katportal users database.

logout(**args*, ***kwargs*)

Logs user out of katportal. Katportal then deletes the cached session_id for this client. In order to call HTTP requests that requires authentication, the user will need to login again.

modify_userlog(**args*, ***kwargs*)

Modify an existing userlog using the dictionary provided as the modified attributes of the userlog.

Parameters

userlog: dict The userlog with the new values to be modified.

tag_ids: list A list of tag id's to link to this userlog. Optional, if this is not specified, the tags attribute of the given userlog will be used. Example: [1, 2, 3, ..]

Returns

userlog: dict The userlog that was modified. Example: {

'other_metadata': [], 'user_id': 1, 'attachments': [], 'tags': '[]', 'timestamp': '2017-02-07 08:47:22', 'start_time': '2017-02-07 00:00:00', 'modified': '', 'content': 'katportalclient userlog modified content!', 'parent_id': '', 'user': {'email': 'cam@ska.ac.za', 'id':

1, 'name': 'CAM'}, 'attachment_count': 0, 'id': 40, 'end_time': '2017-02-07 23:59:59'

}

sb_ids_by_capture_block(**args*, ***kwargs*)

Return list of observation schedule blocks associated with the given capture block ID.

Capture block IDs are provided by SDP and link to the science data archive.

Note: The websocket is not used for this request - it does not need to be connected.

Parameters

capture_block_id: str Capture block identifier. For example: '1556067480'.

Returns

list: List of matching schedule block ID strings. Could be empty.

schedule_block_detail(**args*, ***kwargs*)

Return detailed information about an observation schedule block.

For a list of schedule block IDs, see *schedule_blocks_assigned()*.

Note: The websocket is not used for this request - it does not need to be connected.

Parameters

id_code: str Schedule block identifier. For example: `20160908-0010`.

Returns

dict: Detailed information about the schedule block. Some of the more useful fields are indicated:

```
{ 'description': str,
  'scheduled_time': str,
  'desired_start_time': str,
  'actual_start_time': str,
  'actual_end_time': str,
  'expected_duration_seconds': int,
  'state': str,
  'sub_nr': int,
  ... }

description: str
    Free text description of the observation.
scheduled_time: str
     Time (UTC) at which the Schedule Block went SCHEDULED.
desired_start_time: str
     Time (UTC) at which user would like the Schedule Block␣
↪to start.
actual_start_time: str
     Time (UTC) at which the Schedule Block went ACTIVE.
actual_end_time: str
     Time (UTC) at which the Schedule Block went to␣
↪COMPLETED
```

(continues on next page)

(continued from previous page)

```
    or INTERRUPTED.
expected_duration_seconds: int
    Length of time (seconds) the observation is expected␣
↪to take
    in total.
state: str
    'DRAFT': created, in process of being defined, but not␣
↪yet
             ready for scheduling.
    'SCHEDULED': observation is scheduled for later␣
↪execution, once
                 resources (receptors, correlator, etc.)␣
↪become available.
    'ACTIVE':  observation is currently being executed.
    'COMPLETED': observation completed naturally (may have␣
↪been
                 successful, or failed).
    'INTERRUPTED': observation was stopped or cancelled by␣
↪a user or
                   the system
capture_block_id:
    Capture block identifier set when capture session␣
↪initiates.
    For example: ``1555494792``.
sub_nr: int
    The number of the subarray the observation is scheduled␣
↪on.
```

Raises

ScheduleBlockNotFoundError: If no information was available for the requested schedule block.

schedule_blocks_assigned(**args*, ***kwargs*)

Return list of assigned observation schedule blocks.

The schedule blocks have already been verified and assigned to a single subarray. The subarray queried is determined by the URL used during instantiation. For detail about a schedule block, use *schedule_block_detail()*.

Alternatively, subscribe to a sensor like `sched_observation_schedule_3` for updates on the list assigned to subarray number 3 - see *subscribe()*.

Note: The websocket is not used for this request - it does not need to be connected.

Returns

list: List of scheduled block ID strings. Ordered according to priority of the schedule blocks (first has hightest priority).

Raises

SubarrayNumberUnknown:

- If a subarray number could not be determined.

sensor_detail(**args*, ***kwargs*)
Return detailed attribute information for a sensor.

For a list of sensor names, see `sensors_list()`.

Note: The websocket is not used for this request - it does not need to be connected.

Parameters

sensor_name: str Exact sensor name - see description in *set_sampling_strategy()*.

Returns

dict: Detailed attribute information for the sensor. Some of the more useful fields are indicated:

```
{ 'name': str,
  'description': str,
  'params': str,
  'units': str,
  'type': str,
  'component': str,
  'katcp_name': str,
  ... }

name: str
    Normalised sensor name, as requested in input␣
↪parameters.
description: str
    Free text description of the sensor.
params: str
     Limits or possible states for the sensor value.
units: str
     Measurement units for sensor value, e.g. 'm/s'.
type: str
     Sensor type, e.g. 'float', 'discrete', 'boolean'
component: str
     Name of component that provides the sensor.
katcp_name: str
     Internal KATCP messaging name.
```

Raises

SensorNotFoundError:

- If no information was available for the requested sensor name.
- If the sensor name was not a unique match for a single sensor.

sensor_history(**args*, ***kwargs*)
Return time history of sample measurements for a sensor.

For a list of sensor names, see `sensors_list()`.

Parameters

sensor_name: str Exact sensor name - see description in *set_sampling_strategy()*.

start_time_sec: float Start time for sample history query, in seconds since the UNIX epoch (1970-01-01 UTC).

end_time_sec: float End time for sample history query, in seconds since the UNIX epoch.

include_value_ts: bool Flag to also include value sample_time in addition to time series sample time in the result. Default: False.

timeout_sec: int This parameter is no longer support. Here for backwards compatibility

Returns

list: List of *SensorSample* namedtuples (one per sample, with fields sample_time, value and status) or, if include_value_time was set, then list of *SensorSampleValueTime* namedtuples (one per sample, with fields sample_time, value_time, value and status). See *SensorSample* and *SensorSampleValueTime* for details. If the sensor named never existed, or is otherwise invalid, the list will be empty - no exception is raised.

Raises

SensorHistoryRequestError:

- If there was an error submitting the request.
- If the request timed out

sensor_names(**args*, ***kwargs*)

Return list of matching sensor names.

Provides the list of available sensors in the system that match the specified pattern. For detail about a sensor's attributes, use *sensor_detail()*.

Note: The websocket is not used for this request - it does not need to be connected.

Parameters

filters: str or list of str List of regular expression patterns to match. See *set_sampling_strategies()* for more detail.

Returns

list: List of sensor name strings.

Raises

SensorNotFoundError:

- If any of the filters were invalid regular expression patterns.

sensor_subarray_lookup(**args*, ***kwargs*)

Return full sensor name for generic component and sensor names.

This method gets the full sensor name based on a generic component and sensor name, for a subarray. The subarray queried is determined by the URL used during instantiation. This method will raise an exception if the subarray is not in the 'active' or 'initialising' states.

Note: The websocket is not used for this request - it does not need to be connected.

Parameters

component: str The component that has the sensor to look up.

sensor: str or None The generic sensor to look up. Can be empty or None, in which case just the component is looked up.

katcp_name: bool (optional) True to return the katcp name, False to return the fully qualified Python sensor name. Default is False.

Returns

str: The full sensor name based on the given component and subarray, or just full component name, if no sensor was given.

Raises

SensorLookupError:

- If the requested parameters could not be looked up.

SubarrayNumberUnknown:

- If a subarray number could not be determined.

sensor_value(**args*, ***kwargs*)

Return the latest reading of a sensor.

Note: The websocket is not used for this request - it does not need to be connected.

Parameters

sensor_name: str Exact sensor name. No regular expressions allowed. To get a list of sensor names based off regular expressions, see *sensor_names()*.

components: list List of component names. Default: None

include_value_ts: bool Flag to also include value timestamp. Default: False.

Returns

namedtuple: Instance of *SensorSampleValueTime* if *include_value_ts* is *True*, otherwise an instance of *SensorSample*

Raises

SensorNotFoundError:

- If no information was available for the requested sensor name.

InvalidResponseError:

- When the katportal service returns invalid JSON

sensor_values(**args*, ***kwargs*)

Return a list of latest readings of the sensors matching the specified pattern.

Parameters

filters: str or list of str List of regular expression patterns to match.

e.g. '((m0dd)|(s0ddd))_observer' will return the 'observer' sensor reading for all antennas.

See *set_sampling_strategies()* for more detail.

components: list List of component names Default: None

include_value_ts: bool Flag to also include value timestamp. Default: False.

Returns

dict: Dict of sensor name strings and their latest readings.

Raises

SensorNotFoundError:

- If no information was available for the requested filter.

InvalidResponseError:

- When the katportal service returns invalid JSON

sensors_histories(**args*, ***kwargs*)

Return time histories of sample measurements for multiple sensors.

Finds the list of available sensors in the system that match the specified pattern, and then requests the sample history for each one.

If only a single sensor's data is required, use *sensor_history()*.

Parameters

filters: str or list of str List of regular expression patterns to match. See *set_sampling_strategies()* for more detail.

start_time_sec: float Start time for sample history query, in seconds since the UNIX epoch (1970-01-01 UTC).

end_time_sec: float End time for sample history query, in seconds since the UNIX epoch.

include_value_ts: bool Flag to also include value sample_time in addition to time series sample sample in the result. Default: False.

timeout_sec: int This parameter is no longer support. Here for backwards compatibility

Returns

dict: Dictionary of lists. The keys are the full sensor names. The values are lists of *SensorSample* namedtuples, (one per sample, with fields sample_time, value and status) or, if include_value_time was set, then list of *SensorSampleValueTime* namedtuples (one per sample, with fields sample_time, value_time, value and status). See *SensorSample* and *SensorSampleValueTime* for details.

Raises

SensorHistoryRequestError:

- If there was an error submitting the request.
- If the request timed out

SensorNotFoundError:

- If any of the filters were invalid regular expression patterns.

set_sampling_strategies(**args*, ***kwargs*)
Set up a specified sensor strategy for a filtered list of sensors.

Parameters

namespace: str Namespace with the relevant sensor subscriptions. If empty string '', the general namespace will be used.

filters: str or list of str The regular expression filters to use to select the sensors to which to apply the specified strategy. Use "" to match all sensors. Is matched using KATCP method *list_sensors*. Can be a single string or a list of strings. For example:

1 filter = 'm063_rsc_rxl' 3 filters = ['m063_sensors_ok', 'ap_connected', 'sensors_ok']

strategy_and_params [str] A string with the strategy and its optional parameters specified in space-separated form according the KATCP specification e.g. '<strat_name> <strat_parm1> <strat_parm2>' Examples:

'event' 'period 0.5' 'event-rate 1.0 5.0'

persist_to_redis: bool Whether to persist the sensor updates to redis or not, if persisted to redis, the last updated values can be retrieved from redis without having to wait for the next KATCP sensor update. (default=False)

Returns

dict Dictionary with matching sensor names as keys and the *set_sampling_strategy()* result as value:

```
{ <matching_sensor1_name>:
    { success: bool,
    info: string },
...
<matching_sensorN_name>:
    { success: bool,
    info: string },
}

success: bool
    True if setting succeeded for this sensor, else False.
info: string
    Normalised sensor strategy and parameters as string if
    success == True else, string with the error that␣
↪occured.
```

set_sampling_strategy(**args*, ***kwargs*)
Set up a specified sensor strategy for a specific single sensor.

Parameters

namespace: str Namespace with the relevant sensor subscriptions. If empty string '', the general namespace will be used.

sensor_name: str The exact sensor name for which the sensor strategy should be set. Sensor name has to be the fully normalised sensor name (i.e. python identifier of sensor with all underscores) including the resource the sensor belongs to e.g. 'm063_ap_connected'

strategy_and_params: str A string with the strategy and its optional parameters specified in space-separated form according the KATCP specification e.g. '<strat_name> <strat_parm1> <strat_parm2>' Examples:

'event' 'period 0.5' 'event-rate 1.0 5.0'

persist_to_redis: bool Whether to persist the sensor updates to redis or not, if persisted to redis, the last updated values can be retrieved from redis without having to wait for the next KATCP sensor update. (default=False)

Returns

dict Dictionary with sensor name as key and result as value

sitemap

Returns the sitemap using the URL specified during instantiation.

This method is kept for convenience and backwards compatibility, but should not be used in code that runs on a Tornado event loop as it may block the event loop if the sitemap has not yet been retrieved. Use *get_sitemap()* instead.

sub_nr

Returns subarray number, if available.

This is equivalent to *get_sub_nr()*, but synchronous. It will block the Tornado event loop if the sitemap has not yet been retrieved, so *get_sub_nr()* is preferred.

subscribe(**args*, ***kwargs*)

Subscribe to the specified string identifiers in a namespace.

A namespace provides grouping and consist of channels that can be subscribed to, e.g.

namespace_1 channel_A channel_B

namespace_2 channel_A channel_Z

Messages are then published to namespace channels and delivered to all subscribers.

This method supports both exact string identifiers and redis glob-style pattern string identifiers. Example of glob-style redis patterns:

- h?llo subscribes to hello, hallo and hxllo
- h*llo subscribes to hllo and heeeello
- h[ae]llo subscribes to hello and hallo, but not hillo

Use to escape special characters if you want to match them verbatim.

Parameters

namespace: str Namespace to subscribe to. If an empty string '', the general namespace will be used automatically.

sub_strings: str or list of str The exact and pattern string identifiers to subscribe to. Format = [namespace:]channel. Optional (default='*')

Returns

int Number of strings identifiers subscribed to.

unsubscribe(**args*, ***kwargs*)

Unsubscribe from the specified string identifiers in a namespace.

Method supports both exact string identifiers and redis glob-style pattern string identifiers. For more information refer to the docstring of the *subscribe* method.

Note: Redis requires that the unsubscribe names and patterns must match the original subscribed names and patterns (including any namespaces).

Parameters

namespace: str Namespace to unsubscribe. If an empty string '', the general namespace will be used automatically.

unsub_strings: str or list of str The exact and pattern string identifiers to unsubscribe from. Optional (default='*').

Returns

int Number of strings identifiers unsubscribed from.

userlog_tags(**args*, ***kwargs*)

Return all userlog tags in the database.

Returns

list: List of userlog tags in the database. Example:

[{ 'activated': True, 'slug': '', 'name': 'm047', 'id': 1

}, {

'activated': True, 'slug': '', 'name': 'm046', 'id': 2

}, {

'activated': True, 'slug': '', 'name': 'm045', 'id': 3},

{..}]

userlogs(**args*, ***kwargs*)

Return a list of userlogs in the database that has an start_time and end_time combination that intersects with the given start_time and end_time. For example of an userlog has a start_time before the given start_time and an end time after the given end_time, the time window of that userlog intersects with the time window of the given start_time and end_time.

If an userlog has no end_time, an end_time of infinity is assumed. For example, if the given end_time is after the userlog's start time, there is an intersection of the two time windows.

Here are some visual representations of the time window intersections:

Start End

Userlog: [————-] Search params: [————————]

Start End

Start End

Userlog: [————-] Search params: [————————]

Start End

Start End

Userlog: [————————] Search params: [———]

Start End

Start End

Userlog: [———] Search params: [————————————-]

Start End

Start

Userlog: [————————-* Search params: [————————]

Start End

End

Userlog: *————————————] Search params: [————————]

Start End

Userlog: ———————————— Search params: [————————]

Start End

start_time: str A formatted UTC datetime string used as the start of the time window to query. Format: %Y-%m-%d %H:%M:%S. Default: Today at %Y-%m-%d 00:00:00 (The day of year is selected from local

time but the time portion is in UTC. Example if you are at SAST, and you call this method at 2017-01-01 01:00:00 AM SAST, the date portion of start_time will be selected from local time: 2017-01-01. The start_time is, however, saved as UTC, so this default will be 2017-01-01 00:00:00 AM UTC and NOT 2016-12-31 00:00:00 AM UTC)

end_time: str A formatted UTC datetime string used as the end of the time window to query. Format: %Y-%m-%d %H:%M:%S. Default: Today at %Y-%m-%d 23:59:59 (The day of year is selected from local

time but the time portion is in UTC. Example if you are at SAST, and you call this method at 2017-01-01 01:00:00 AM SAST, the date portion of end_time will be selected from local time: 2017-01-01. The end_time is, however, saved as UTC, so this default will be 2017-01-01 23:59:59 UTC and NOT 2016-12-31 23:59:59 UTC)

Returns

list: List of userlog that intersects with the give start_time and end_time. Example:

[{

'other_metadata': [], 'user_id': 1, 'attachments': [], 'tags': '[]', 'timestamp': '2017-02-07 08:47:22', 'start_time': '2017-02-07 00:00:00', 'modified': '', 'content': 'katportalclient userlog creation content!', 'parent_id': '', 'user': {'email': 'cam@ska.ac.za', 'id': 1, 'name': 'CAM'}, 'attachment_count': 0, 'id': 40, 'end_time': '2017-02-07 23:59:59'

}, {..}]

exception katportalclient.client.**ScheduleBlockNotFoundError**

Bases: exceptions.Exception

Raise if requested schedule block is not found.

exception katportalclient.client.**ScheduleBlockTargetsParsingError**

Bases: exceptions.Exception

Raise if there was an error parsing the targets attribute of the ScheduleBlock

exception katportalclient.client.**SensorHistoryRequestError**
Bases: exceptions.Exception

Raise if error requesting sensor sample history.

exception katportalclient.client.**SensorLookupError**
Bases: exceptions.Exception

Raise if requested sensor lookup failed.

exception katportalclient.client.**SensorNotFoundError**
Bases: exceptions.Exception

Raise if requested sensor is not found.

class katportalclient.client.**SensorSample**
Bases: *katportalclient.client.SensorSample*

Class to represent all sensor samples.
Fields:

- **sample_time: float** The timestamp (UNIX epoch) the sample was received by CAM. timestamp value is reported with at least millisecond precision.
- **value: str** The value of the sensor when sampled. The units depend on the sensor, see *sensor_detail()*.
- **status: str** The status of the sensor when the sample was taken. As defined by the KATCP protocol. Examples: 'nominal', 'warn', 'failure', 'error', 'critical', 'unreachable', 'unknown', etc.

Methods

csv(self)	Returns sample in comma separated values format.

csv(*self*)
Returns sample in comma separated values format.

class katportalclient.client.**SensorSampleValueTime**
Bases: *katportalclient.client.SensorSampleValueTime*

Class to represent sensor samples, including the value_time.
Fields:

- **sample_time: float** The timestamp (UNIX epoch) the sample was received by CAM. Timestamp value is reported with at least millisecond precision.
- **value_time: float** The timestamp (UNIX epoch) the sample was read at the lowest level sensor. value_timestamp value is reported with at least millisecond precision.
- **value: str** The value of the sensor when sampled. The units depend on the sensor, see *sensor_detail()*.
- **status: str** The status of the sensor when the sample was taken. As defined by the KATCP protocol. Examples: 'nominal', 'warn', 'failure', 'error', 'critical', 'unreachable', 'unknown', etc.

Methods

csv(self)	Returns sample in comma separated values format.

csv(*self*)
Returns sample in comma separated values format.

exception katportalclient.client.**SubarrayNumberUnknown**
Bases: exceptions.Exception

Raised when subarray number is unknown

katportalclient.client.**create_jwt_login_token**(*email*, *password*)
Creates a JWT login token. See http://jwt.io for the industry standard specifications.

Parameters

email: str The email address of the user to include in the token. This email address needs to exist in the kaportal user database to be able to authenticate.

password: str The password for the user specified in the email address to include in the JWT login token.

Returns

jwt_auth_token: str The authentication token to include in the HTTP Authorization header when verifying a user's credentials on katportal.

4.1.2 request

Module defining the JSON-RPC request class used by websocket client.

class katportalclient.request.**JSONRPCRequest**(*method*, *params*)
Bases: future.types.newobject.newobject

Class with structure following the JSON-RPC standard.

Parameters

method: str

Name of the remote procedure to call.

params: list List of parameters to be used for the remote procedure call.

Attributes

params

Methods

__call__(self)	Return object's attribute dictionary in JSON form.
method_and_params_hash(self)	Return a hash for the methods and params attributes for easy comparison

id = ''

method = ''

method_and_params_hash(*self*)
Return a hash for the methods and params attributes for easy comparison

```
    params = None
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

U