developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Aug 11, 2021

HOME

The OET REST server implements the services described in *REST API*.

Direct communication with the server via http is described in *REST API*. User CLI communication with the server is facilitated by the *REST Client*.

`make rest` starts the OET REST server.

CHAPTER

ONE

REST CLIENT

SKA observations will be controlled by 'Procedures'. Each 'Procedure' comprises a Python script and a set of arguments, some of which will be set when the script is loaded and some at run-time.

The management of 'Procedures' and the processes which execute them is handled by the *REST Server*, which implements the methods described in the *REST API*. The *REST Server* lets the user:

- Load requested Procedure scripts with initialization arguments and have them ready for execution.
- When required, pass run-time arguments to a script and start a process executing it.
- Stop the script mid-execution by terminating the process executing it.

The REST Client provides a command line interface (CLI) through which the user can communicate with the *REST Server* remotely. The address of the remote REST server can be specified at the command line via the `server-url` argument, or set session-wide by setting the `OET_REST_URI` environment variable, e.g.,:

```
export OET_REST_URI=http://my-rest-service:5000/api/v1.0/procedures
```

By default, the client assumes it is operating within a SKAMPI environment and attempts to connect to a REST server using the default REST service name of http://oet-rest:5000/api/v1.0/procedures. If running the OET client within SKAMPI via the oet-ssh or oet-jupyter services, the `OET_REST_URI` variable is automatically set.

The methods available through the REST Client map closely to the *REST API* of the server and are described below.

REST Client Method	Parameters	Default	Description
create	server-url	See note above	**Prepare a new procedure** Load the requested script and prepare it for execution. Arguments provided here are passed to the script init function, if defined OET maintains record of 10 newest scripts which means creating 11th script will remove the oldest script from the record.
	script-uri	None	
	args	None	
	kwargs	–subarray_id=1	
list	server-url	See note above	**List procedures**
	pid	None	Return info on the collection of 10 newest procedures, or info on the one specified by process ID (pid)
start	server-url	See note above	**Start a Procedure Executing** Start a process executing the procedure specified by process ID (pid) or, if none is specified start the last one loaded. Only one procedure can be executing at any time
	pid	None	
	args	None	
	kwargs	None	
stop	server-url	See note above	**Stop Procedure Execution** Stop a running process executing the procedure specified by process ID (pid) or, if none is specified, stop the currently running process. If run_abort flag is True, OET will send Abort command to the SubArray as part of script termination.
	pid	None	
	run_abort	True	
describe	server-url	See note above	**Investigate a procedure** Displays the call arguments, state history and, if the procedure failed, the stack trace of a specified process ID (pid). If no pid is specified describe the last process created.
	pid	None	
Listen	server-url	http://oet-rest:5000/api/v1.0/stream	**Get real times scripts events** Get a real time delivery of events published by oet scripts

In the table 'args' refers to parameters specified by position on the command line, 'kwargs' to those specified by name e.g. –myparam=12.

1.1 Help Information

General help information can be obtained by typing the command:

```
$ oet
```

Detailed help information for specific commands is also available e.g.:

```
$ oet create --help
```

1.2 Examples

This section runs through an example session in which we will load two new 'Procedures' and then run one of them. First we load the procedures:

```
$ oet create file://test.py 'hello' --verbose=true
```

which will generate the output:

```
  ID  Script           Creation time        State
----  ---------------  -------------------  -------
   1  file://test.py   2020-09-30 10:30:12  CREATED
```

Note the use of both positional and keyword/value arguments for the procedure on the command line. Now create a second procedure:

```
$ oet create file://test2.py 'goodbye'
```

giving:

```
 ID   Script           Creation time        State
----  ---------------  -------------------  -------
  2  file://test2.py  2020-09-30 10:35:12  CREATED
```

We can check the state of the procedures currently loaded by:

```
$ oet list
```

giving:

```
 ID   Script           Creation time        State
----  ---------------  -------------------  -------
   1  file://test.py   2020-09-30 10:30:12  CREATED
   2  file://test2.py  2020-09-30 10:35:12  CREATED
```

Alternatively, we could check the state of procedure 2 by typing:

```
$ oet list --pid=2
```

giving:

```
 ID   Script           Creation time        State
----  ---------------  -------------------  -------
  2   file://test2.py  2020-09-30 10:35:12  CREATED
```

Now that we have our procedures loaded we can start one of them running. At this point we supply the index number of the procedure to run, and some runtime arguments to pass to it if required.

```
$ oet start --pid=2 'bob' --simulate=false
```

giving:

```
  ID   Script           Creation time        State
----  ---------------  -------------------  -------
  2   file://test2.py  2020-09-30 10:35:12  RUNNING
```

A 'list' command will give the same information:

```
$ oet list
```

giving:

```
  ID   Script           Creation time        State
----  ---------------  -------------------  -------
   1  file://test.py   2020-09-30 10:30:12  CREATED
   2  file://test2.py  2020-09-30 10:35:12  RUNNING
```

A 'describe' command will give further detail on a procedure, no matter its state.:

```
$oet describe --pid=2
```

giving:

```
  ID  Script           URI
----  ---------------  ----------------------------------------
   2  file://test2.py  http://0.0.0.0:5000/api/v1.0/procedures/2

Time                        State
--------------------------  -------
2020-09-30 10:19:38.646475  CREATED
2020-09-30 10:35:12.605270  RUNNING

Method    Arguments    Keyword Arguments
--------  -----------  -------------------
init      []           {'subarray_id': 1}
run       []           {}
```

If the procedure failed, then the stack trace will also be displayed.

A 'listen' command will give the real time delivery of oet events published by scripts:

```
$ oet listen
```

giving:

```
event: request.procedure.list
data: args=() kwargs={'msg_src': 'FlaskWorker', 'request_id': 1604056049.4846392, 'pids
↪': None}

event: procedure.pool.list
data: args=() kwargs={'msg_src': 'SESWorker', 'request_id': 1604056049.4846392, 'result
↪': []}

event: request.procedure.create
data: args=() kwargs={'msg_src': 'FlaskWorker', 'request_id': 1604056247.0666442, 'cmd':␣
↪PrepareProcessCommand(script_uri='file://scripts/eventbus.py', init_args=
↪<ProcedureInput(, subarray_id=1)>)}
```

(continues on next page)

(continued from previous page)

```
event: procedure.lifecycle.created
data: args=() kwargs={'msg_src': 'SESWorker', 'request_id': 1604056247.0666442, 'result
↪': ProcedureSummary(id=1, script_uri='file://scripts/eventbus.py', script_args={'init
↪': <ProcedureInput(, subarray_id=1)>, 'run': <ProcedureInput(, )>}, history=
↪<ProcessHistory(process_states=[(ProcedureState.CREATED, 1604056247.713874)],␣
↪stacktrace=None)>, state=<ProcedureState.CREATED: 1>)}
```

1.3 Example session in a SKAMPI environment

From a shell, you can use the 'oet' command to trigger remote execution of a full observation, e.g.,:

```
# create process for telescope start-up and execute it
oet create file:///scripts/startup.py
oet start

# create process for resource allocation script
oet create file:///scripts/allocate_from_file_sb.py --subarray_id=3
oet start scripts/example_sb.json

# create process for configure/scan script
oet create file:///scripts/observe_sb.py --subarray_id=3
# run the script, specifying scheduling block JSON which defines
# the configurations, and the order and number of scans
oet start scripts/example_sb.json

# create process for resource deallocation script
oet create file:///scripts/deallocate.py --subarray_id=3
# run with no arguments, which requests deallocation of all resources
oet start

# create process for telescope standby script
oet create file:///scripts/standby.py
oet start
```

CHAPTER

TWO

REST API

A 'Procedure' represents a script along with its load-time arguments and runtime arguments. The REST API operates on procedures.

The workflow for the script execution service is to:

- Load a requested Python script(s) ready for execution;
- When requested, start execution of the requested script;
- Abort the script mid-execution if requested with an option to send the abort command to sub-array.

This workflow has been mapped to the following REST API:

HTTP Method	Resource URL	Description
GET	/api/v1/procedures	**List procedures**: Return the collection of all prepared and running procedures
GET	/api/v1/procedures/<id>	**Return a procedure definition**
GET	/api/v1/stream	**Streaming real time oet events**: Return a real time oet events published by scripts
POST	/api/v1/procedures	**Prepare a new procedure** Loads the requested script and prepares it for execution
PUT	/api/v1/procedures/<id>	**Modify a procedure** Modifies the state of a prepared procedure. This can be used to start execution by setting the 'state' procedure attribute to `RUNNING` or stop execution by setting 'state' to `STOPPED`.

Procedures are defined as JSON objects with the following fields:

Field	JSON Type	Description
uri	string	Read-only procedure URI. Defined by server on procedure creation.
script_uri	string	URI of the script to execute, e.g., file:///path/to/obsscript.py
script_args	object	**TO BE REFINED!** Dict of input arguments to provide to methods in the script. Only two methods are recognised at the moment: 'init', called at script creation time, and 'run', called to commence script execution. Keys are the name of the script method, values are dicts with two keys ('args' and 'kwargs') for positional arguments and keyword/value arguments respectively. For example, below represents a call to init(1,2,3,subarray_id=1): `"script_args": { "init": { "args": [ 1, 2, 3 ], "kwargs": { "subarray_id": "1" } } }`
state	str	Script execution state: CREATED, RUNNING, STOPPED, COMPLETED, FAILED.
history	object	history contains a Dict of process_states and stacktrace. process_states which contains a Dict of ProcedureStates and timestamps for each state (e.g. {'CREATED': 18392174.543, 'RUNNING': 18392143.546, 'COMPLETED': 183925456.744}). stacktrace which is None on default and will be populated with the stacktrace from the script if script execution raises an exception.

Below is a JSON representation of an example procedure resource. This resource (located at URI http://localhost:5000/api/v1.0/procedures/1), represents a script (located on disk at /path/to/observing_script.py), that has been loaded and its initialisation method called with two arguments (e.g, the script init function was called as `init(subarray_id=1, sb_uri='file:///path/to/scheduling_block_123.json')`. The script is ready to execute but is not yet executing, as shown by its state being *CREATED* `:

```
{
    "script_args": {
      "init": {
        "args": [],
        "kwargs": {
          "sb_uri": "file:///path/to/scheduling_block_123.json",
          "subarray_id": 1
        }
      },
      "run": {
        "args": [],
        "kwargs": {}
      }
    },
    "script_uri": "file:///path/to/observing_script.py",
    "history": {
        "process_states": {
             "CREATED": 1601463545.7789776
            },
        "stacktrace": null
        },
    "state": "CREATED",
    "uri": "http://localhost:5000/api/v1.0/procedures/1"
}
```


2.1 Examples

The following examples show some interactions with the REST service from the command line, using curl to send input to the service and with responses output to the terminal.

2.1.1 Creating a procedure

The session below creates a new procedure, which loads the script and calls the script's init() function, but does not commence execution. The created procedure is returned as JSON. Note that in the return JSON the procedure URI is defined. This URI can be used in a PUT request that commences script execution:

```
tangodev@buster:~/ska/ska-oso-oet$ curl -i -H "Content-Type: application/json" -X POST -
↪d '{"script_uri":"file:///path/to/observing_script.py", "script_args": {"init": {
↪"kwargs": {"subarray_id": 1, "sb_uri": "file:///path/to/scheduling_block_123.json"} } }
↪}' http://localhost:5000/api/v1.0/procedures
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 424
Server: Werkzeug/0.16.0 Python/3.7.3
Date: Wed, 15 Jan 2020 10:08:01 GMT

{
  "procedure": {
    "script_args": {
      "init": {
```

(continues on next page)

(continued from previous page)


```
        "args": [],
        "kwargs": {
          "sb_uri": "file:///path/to/scheduling_block_123.json",
          "subarray_id": 1
        }
      },
      "run": {
        "args": [],
        "kwargs": {}
      }
    },
    "script_uri": "file:///path/to/observing_script.py",
    "history": {
        "process_states": {
            "CREATED": 1601463545.7789776
            },
        "stacktrace": null
    },
    "state": "CREATED",
    "uri": "http://localhost:5000/api/v1.0/procedures/2"
  }
}
```


2.1.2 Listing all procedures

The session below lists all procedures, both running and non-running. This example shows two procedures have been created: procedure #1 that will run resource_allocation.py, and procedure #2 that will run observing_script.py:


```
tangodev@buster:~/ska/ska-oso-oet$ curl -i http://localhost:5000/api/v1.0/procedures
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 913
Server: Werkzeug/0.16.0 Python/3.7.3
Date: Wed, 15 Jan 2020 10:11:42 GMT

{
  "procedures": [
    {
      "script_args": {
        "init": {
          "args": [],
          "kwargs": {
            "dishes": [
              1,
              2,
              3
            ]
          }
        },
        "run": {
          "args": [],
```


(continues on next page)

(continued from previous page)


```
          "kwargs": {}
        }
      },
      "script_uri": "file:///path/to/resource_allocation.py",
      "history": {
          "process_states": {
               "CREATED": 1601463545.7789776
            },
          "stacktrace": null
             },
      "state": "CREATED",
      "uri": "http://localhost:5000/api/v1.0/procedures/1"
    },
    {
      "script_args": {
        "init": {
          "args": [],
          "kwargs": {
            "sb_uri": "file:///path/to/scheduling_block_123.json",
            "subarray_id": 1
          }
        },
        "run": {
          "args": [],
          "kwargs": {}
        }
      },
      "script_uri": "file:///path/to/observing_script.py",
      "history": {
           "process_states": {
               "CREATED": 1601463545.7789885
              },
           "stacktrace": null
      },
      "state": "CREATED",
      "uri": "http://localhost:5000/api/v1.0/procedures/2"
    }
  ]
}
```


2.1.3 Listing one procedure

A specific procedure can be listed by a GET request to its specific URI. The session below lists procedure #1:

```
tangodev@buster:~/ska/ska-oso-oet$ curl -i http://localhost:5000/api/v1.0/procedures/1
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 417
Server: Werkzeug/0.16.0 Python/3.7.3
Date: Wed, 15 Jan 2020 10:18:26 GMT
```

(continues on next page)

(continued from previous page)

```
{
  "procedure": {
    "script_args": {
      "init": {
        "args": [],
        "kwargs": {
          "dishes": [
            1,
            2,
            3
          ]
        }
      },
      "run": {
        "args": [],
        "kwargs": {}
      }
    },
    "script_uri": "file:///path/to/resource_allocation.py",
    "history": {
        "process_states": {
            "CREATED": 1601463545.7789776
            },
        "stacktrace": null
    },
    "state": "CREATED",
    "uri": "http://localhost:5000/api/v1.0/procedures/1"
  }
}
```

2.1.4 Starting procedure execution

The signal to begin script execution is to change the state of a procedure to `RUNNING`. This is achieved with a PUT request to the resource. Any additional late-binding arguments to pass to the script's run() function should be defined in the 'run' script_args key.

The example below requests execution of procedure #2, with late binding kw argument scan_duration=14:

```
tangodev@buster:~/ska/ska-oso-oet$ curl -i -H "Content-Type: application/json" -X PUT -d
↪'{"script_args": {"run": {"kwargs": {"scan_duration": 14.0}}}, "state": "RUNNING"}'␣
↪http://localhost:5000/api/v1.0/procedures/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 467
Server: Werkzeug/0.16.0 Python/3.7.3
Date: Wed, 15 Jan 2020 10:14:06 GMT

{
  "procedure": {
    "script_args": {
      "init": {
```

(continues on next page)

(continued from previous page)

```
        "args": [],
        "kwargs": {
          "sb_uri": "file:///path/to/scheduling_block_123.json",
          "subarray_id": 1
        }
      },
      "run": {
        "args": [],
        "kwargs": {
          "scan_duration": 14.0
        }
      }
    },
    "script_uri": "file:///path/to/observing_script.py",
    "history": {
        "process_states": {
            "CREATED": 1601463545.7789885,
            "RUNNING": 1601463545.7789997
         },
        "stacktrace": null
    }
    "state": "RUNNING",
    "uri": "http://localhost:5000/api/v1.0/procedures/2"
  }
}
```

2.1.5 Aborting process execution

The signal to abort script mid-execution is to change the state of a procedure to `STOPPED`. This is achieved with a PUT request to the resource. Additional argument *abort* can be provided in the request which, when true, will execute an abort script that will send Abort command to the sub-array device. The default value of *abort* is False.

```
tangodev@buster:~/ska/ska-oso-oet$ curl -i -H "Content-Type: application/json" -X PUT -d
↪'{"abort": true, "state": "STOPPED"}' http://localhost:5000/api/v1.0/procedures/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 467
Server: Werkzeug/0.16.0 Python/3.7.3
Date: Wed, 15 Jan 2020 10:14:09 GMT
{"abort_message":"Successfully stopped script with ID 2 and aborted subarray activity "}
```

2.1.6 When an error occurs

If there is a mistake in the User input it is desirable that the API produces errors in a consistent computer-readable way.

The session below attempts to list a procedure which does not exist:

```
tangodev@buster:~/ska/ska-oso-oet$ curl -i http://localhost:5000/api/v1.0/procedures/4
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 103
Server: Werkzeug/1.0.1 Python/3.7.3
Date: Thu, 18 Feb 2021 17:40:30 GMT

{"error": "404 Not Found", "type": "ResourceNotFound", "Message": "No information␣
↪available for PID=4"}
```

2.1.7 Listen real time oet events

The session below lists all events published by oet scripts. This example shows two events, #1 request to available procedures #2 get the details of all the created procedures

```
tangodev@buster:~/ska/ska-oso-oet$ curl -i http://localhost:5000/api/v1.0/stream
HTTP/1.0 200 OK
Content-Type: text/event-stream; charset=utf-8
Connection: close
Server: Werkzeug/1.0.1 Python/3.7.3
Date: Mon, 02 Nov 2020 06:57:40 GMT

data:{"msg_src": "FlaskWorker", "pids": null, "topic": "request.procedure.list"}
id:1605017762.46912

data:{"msg_src": "SESWorker", "result": [], "topic": "procedure.pool.list"}
id:1605017762.46912

data:{"msg_src": "FlaskWorker", "cmd": {"py/object": "oet.procedure.application.
↪application.PrepareProcessCommand", "script_uri": "file://scripts/eventbus.py", "init_
↪args": {"py/object": "oet.procedure.domain.ProcedureInput", "args": {"py/tuple": []},
↪"kwargs": {"subarray_id": 1}}}, "topic": "request.procedure.create"}
id:1605017784.1536236

data:{"msg_src": "SESWorker", "result": {"py/object": "oet.procedure.application.
↪application.ProcedureSummary", "id": 1, "script_uri": "file://scripts/eventbus.py",
↪"script_args": {"init": {"py/object": "oet.procedure.domain.ProcedureInput", "args": {
↪"py/tuple": []}, "kwargs": {"subarray_id": 1}}, "run": {"py/object": "oet.procedure.
↪domain.ProcedureInput", "args": {"py/tuple": []}, "kwargs": {}}}, "history": {"py/
↪object": "oet.procedure.domain.ProcedureHistory", "process_states": {"py/reduce": [{
↪"py/type": "collections.OrderedDict"}, {"py/tuple": []}, null, null, {"py/tuple": [{
↪"py/tuple": [{"py/reduce": [{"py/type": "oet.procedure.domain.ProcedureState"}, {"py/
↪tuple": [1]}]}, 1605017786.0569353]}]}]}, "stacktrace": null}, "state": {"py/id": 5}},
↪"topic": "procedure.lifecycle.created"}
id:1605017784.1536236
```

CHAPTER

THREE

OBSERVATION EXECUTION TOOL

3.1 Project description

The *ska-oso-oet* project contains the code for the Observation Execution Tool (OET), the application which provides on-demand Python script execution for the SKA.

3.2 Overview

The OET consists of a script execution engine, which loads specified scripts and runs them in child Python processes, and a REST layer which makes the API for the script execution engine available via REST over HTTP.

The REST layer is made up of two components that work together to provide the remote script execution functionality:

- The OET *REST Server* maintains a list of the scripts that have been loaded and their current state. The server implements the interface specified by the OET *REST API*.
- The OET *REST Client* provides a Command Line Interface (CLI) to the OET *REST Server*. It does this by translating and communicating HTTP messages to and from the server.

Note: SKA control scripts are not packaged as part of this project. The repository of observing scripts executed by the OET can be found in the OET Scripts project.

3.3 Quickstart

This project is structured to use Docker containers for development and testing so that the build environment, test environment and test results are all completely reproducible and are independent of host environment. It uses `make` to provide a consistent UI (see *Makefile targets*).

Build a new Docker image for the OET with:

```
make build
```

Execute the test suite and lint the project with:

```
make test
make lint
```

3.4 Makefile targets

This project contains a Makefile which acts as a UI for building Docker images, testing images, and for launching interactive developer environments. The following make targets are defined:

Makefile target	Description
build	Build a new application image
test	Test the application image
lint	Lint the application image
prune	Delete stale Docker images for this project
interactive	Launch a minimal Tango system (including the device under development), mounting the source directory from the host machine inside the container
push	Push the application image to the Docker registry
up	launch the development/test container service on which this application depends
down	stop all containers launched by 'make up' and 'make interactive'
rest	start the OET REST server
help	show a summary of the makefile targets above

3.4.1 Creating a new application image

`make build` target creates a new Docker image for the application based on the 'ska-python-runtime' image. To optimise final image size and to support the inclusion of C extension Python libraries such as pytango, the application is built inside an intermediate Docker image which includes compilers and cached eggs and wheels for commonly-used Python libraries ('ska-python-builder'). The resulting Python environment from this intermediate stage is copied into a final image which extends a minimal SKA Python runtime environment ('ska-python-runtime'), to give the final Docker image for this application.

3.4.2 Interactive development using containers

`make interactive` launches an interactive session using the application image, mounting the project source directory at /app inside the container. This allows the container to run code from the local workspace. Any changes made to the project source code will immediately be seen inside the container.

3.4.3 Test execution

`make test` runs the application test procedure defined in test-harness/Makefile in a temporary container. The Makefile example for this project runs 'python setup.py test' and copies the resulting output and test artefacts out of the container and into a 'build' directory, ready for inclusion in the CI server's downloadable artefacts.

3.4.4 REST server

`make rest` starts the OET REST server. Details of the REST API can be found in *REST API*. Instructions on how to use the REST client can be found here: *REST Client*.

3.4.5 Feature flags

OET feature flags are configured via environment variables and configuration files. The configuration file, oet.ini, can be located either in the user's home directory, or the root of the installation folder.

Feature flags are read in this order:

1. environment variable;
2. oet.ini configuration file;
3. default flag value as specified in OET code.

Available feature flags are:

environment variable	oet.ini setting	Type	Description	Default
OET_READ_VIA_PUBSUB	read_via_pubsub	Boolean	sets whether pubsub or the alternative, polling, is used to read from tango	False