developer.skatelescope.org Documentation

Release 5.2.0

Marco Bartolini

Aug 07, 2023

HOME

The `oet` command can be used to control a remote OET deployment[1]. The `oet` command has two sub-commands, `procedure` and `activity`.

`oet procedure` commands are used to control individual observing scripts, which includes loading and starting and stopping script execution.

`oet activity` commands are used to execute more general activities on the telescope, for example running the allocate activity on SB with ID xxx.

See *Procedure* and *Activity* sections for further details on commands available for each of the approaches.

General help and specific help is available at the command line by adding the `--help` argument. For example:

```
# get a general overview of the OET CLI
$ oet procedure --help
$ oet activity --help

# get specific help on the oet create command
$ oet procedure create --help

# get specific help on the oet describe command
$ oet activity describe --help
```

[1] Specifically, the cli tool acts as a REST client that interfaces with the OET REST API described in *Module View: REST API*.

CHAPTER

ONE

INSTALLATION

The OET command line tool is available as the `oet` command at the terminal. The OET CLI is packaged separately so it can be installed without OET backend dependencies, such as PyTango. It can be installed into a Python environment, and configured to access a remote OET deployment as detailed below:

```
$ pip install --upgrade ska_oso_oet_client
```

By default, the OET image has the CLI installed, meaning the CLI is accessible from inside the running OET pod.

CHAPTER

TWO

CONFIGURATION

The address of the remote OET backend 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.,

```
# provide the server URL when running the command, e.g.
$ oet --server-url=http://my-oet-deployment.com:5000/api/v1.0 procedure list

# alternatively, set the server URL for a session by defining an environment variable
$ export OET_REST_URI=http://my-oet-deployment.com:5000/api/v1.0
$ oet procedure list
$ oet activity describe
$ oet procedure create ...
```

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://ska-oso-oet-rest:5000/api/v1.0. If running the OET client within a SKAMPI pod, the `OET_REST_URI` should automatically be set.

CHAPTER

THREE

COMMANDS

3.1 Common

The `oet` CLI tool has `listen` command which is neither activity or procedure specific. It is used to observe OET messages and script messages from, procedure, activity and several other topics.

OET CLI action	Parameters	Default	Description
Listen	server-url	See *Configuration* section	**Get real times scripts events** Get a real time delivery of events published by oet server/scripts

3.1.1 Examples

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

```
$ oet listen

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: activity.pool.list
data: args=() kwargs={'msg_src': 'ActivityWorker', 'request_id': 1604056078.4847652,
↪'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)>)}

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
```

(continues on next page)

(continued from previous page)

```
↪': <ProcedureInput(, subarray_id=1)>, 'run': <ProcedureInput(, )>}, history=
↪<ProcessHistory(process_states=[(ProcedureState.READY, 1604056247.713874)],␣
↪stacktrace=None)>, state=<ProcedureState.READY: 1>)}
```

Press `Control-c` to exit from `oet listen`.

3.2 Procedure

Using `oet procedure`, a remote OET deployment can be instructed to:

1. load a Python script using `oet procedure create`;
2. run a function contained in the Python script using `oet procedure start`;
3. stop a running Python function using `oet procedure stop`;

In addition, the current and historic state of Python processes running on the backend can be inspected with

1. `oet procedure list` to list all scripts that are prepared to run or are currently running;
2. `oet procedure describe` to inspect the current and historic state of a specific process.

The commands available via `oet procedure` are described below.

OET CLI action	Parameters	Default	Description
create	server url	See *Configuration* section	**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	
	kwarg	--subarray_id=1 --git_repo= "http://gitlab.com/ska-telescope/oso/ska-oso-scripting" --git_branch="master" --git_commit=None --create_env=False	
list	server url	See *Configuration* section	**List procedures** Return info on the collection of 10 newest procedures, or info on the one specified by process ID (pid)
	pid	None	
start	server url	See *Configuration* section	**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. listen flag is set to True by default which means that events are shown on the command line unless is is explicitly set to False.
	pid	None	
	args	None	
	kwarg	None	
	listen	True	
stop	server url	See *Configuration* section	**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_a	True	
describe	server url	See *Configuration* section	**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	

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

3.2.1 Examples

This section runs through an example session in which we will load two new 'Procedures'[2] and then run one of them. First we load the procedure, and see the backend report that it is creating a process with ID=1 to run the script.

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

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

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

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

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

Now create a third procedure that will be pulled from git:

```
$ oet procedure create git://test3.py --git_repo="http://foo.git" --git_branch="test" --
↪create_env=True

 ID   Script           Creation time        State
----  ---------------  -------------------  -------
  3  git://test3.py    2020-09-30 10:40:12  CREATING
```

We can check the state of the procedures currently loaded:

```
$ oet procedure list

 ID   Script           Creation time        State
----  ---------------  -------------------  -------
   1  file://test.py   2020-09-30 10:30:12  READY
   2  file://test2.py  2020-09-30 10:35:12  READY
   3  git://test3.py   2020-09-30 10:40:12  READY
```

Alternatively, we could check the state of procedure 2 alone:

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

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

Now that we have our procedures loaded we can start one of them running. At this point we supply the ID of the procedure to run, and some runtime arguments to pass to it if required. The backend responds with the new status of the procedure.

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

(continues on next page)

[2] For reference, the OET architecture refers to Python scripts as *Procedures*.

(continued from previous page)

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

An `oet procedure list` command also shows the updated status of procedure #2:

```
$ oet procedure list

  ID   Script           Creation time        State
----  ---------------  -------------------  -------
   1  file://test.py   2020-09-30 10:30:12  READY
   2  file://test2.py  2020-09-30 10:35:12  RUNNING
   3  git://test3.py   2020-09-30 10:40:12  READY
```

An `oet procedure describe` command will give further detail on a procedure, no matter its state.

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

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

 Time                        State
 --------------------------  -------
 2020-09-30 10:19:38.011584  CREATING
 2020-09-30 10:19:38.016266  IDLE
 2020-09-30 10:19:38.017883  LOADING
 2020-09-30 10:19:38.018880  IDLE
 2020-09-30 10:19:38.019006  RUNNING 1
 2020-09-30 10:19:38.019021  READY
 2020-09-30 10:35:12.605270  RUNNING 2

 Index     Method     Arguments    Keyword Arguments
 --------  ---------  -----------  -------------------
     1      init      ['goodbye']  {'subarray_id': 1}
     2      run       ['bob']      {'simulate': false}
```

Describing a script from git shows additional information on the repository:

```
$ oet procedure describe --pid=3

   ID  Script           URI
 ----  ---------------  ----------------------------------------
    3  git://test3.py    http://0.0.0.0:5000/api/v1.0/procedures/3

 Time                        State
 --------------------------  -------
 2020-09-30 10:40:12.435305  CREATING
 2020-09-30 10:40:12.435332  IDLE
 2020-09-30 10:40:12.435364  LOADING
 2020-09-30 10:40:12.435401  IDLE
 2020-09-30 10:40:12.435433  RUNNING 1
 2020-09-30 10:40:12.435642  READY
```

(continues on next page)

(continued from previous page)

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

 Repository           Branch    Commit
 ---------------      -------   -------------------
 http://foo.git       test
```

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

3.2.2 Example session in a SKAMPI environment

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

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

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

# create process for configure/scan script
oet procedure 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 procedure start scripts/example_sb.json

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

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

3.3 Activity

Using `oet activity`, a remote OET deployment can be instructed to:

1. execute a observing activity of a Scheduling Block with `oet activity run`

In addition, the current and historic state of Activities can be inspected with

1. `oet activity list` to list all activities that have been started;
2. `oet activity describe` to inspect the current and historic state of a specific activity.

The commands available via `oet activity` are described below.

OET CLI action	Parameters	Default	Description
run	server url	See *Configuration* section	**Run an activity of an SB** Create and run a script referenced by an activity defined in an SB. The activity-name and sbd-id are mandatory arguments. script-args is a dictionary defining function name as a key (e.g. 'init') and any keyword arguments to be passed for the function on top of arguments present in the SB. Only keyword args are currently allowed. preparep-only should be set to False if the script referred to by SB and activity is not to be run yet. To start a prepared script, use the *oet procedure* commands. create-env flag should be set to True if script referred to by SB is a Git script and requires a non- default environment to run.
	activi name	None	
	sbd-id	None	
	script args	None	
	prepa only	False	
	creat env	False	
	listen	True	
list	server url	See *Configuration* section	**List activities** Return info on the collection of 10 newest activities, or info on the one specified by activity ID (aid)
	aid	None	
describe	server url	See note above	**Investigate an activity** Displays the script arguments, and the state history of a specified activity ID (aid). If no aid is specified describe the last activity created.
	aid	None	

3.3.1 Examples

This section runs through an example session in which we will run an activity with arguments to the script. We will also demonstrate the more advanced use of controlling activity execution with additional `oet procedure` commands. For this we will prepare an activity without executing it and use the `oet procedure` commands to run the prepared activity.

```
$ oet activity run allocate sbd-123 --script-args='{"init": {"kwargs": {"foo": "bar"}}}'

  ID  Activity    SB ID    Creation Time          Procedure ID  State
----  ----------  -------  -------------------  --------------  ---------
   1  allocate    sbd-123  2023-01-06 13:56:47               1  REQUESTED
```

Note the use of keyword arguments for the script arguments. These will be passed as arguments when each function in the script is run. If the given keyword argument is already defined in the Scheduling Block, the value will be overwritten with the user provided one.

The activity has now been started and will complete without any further interaction from the user.

For an example of more advanced use of the activity interface, run an activity but set the `prepare-only` flag to True:

```
$ oet activity run observe sbd-123 --prepare-only=True

  ID  Activity    SB ID    Creation Time          Procedure ID  State
----  ----------  -------  -------------------  --------------  ---------
   2  observe     sbd-123  2023-01-06 13:56:56               2  REQUESTED
```

We can check the state of the activities currently present:

```
$ oet activity list

  ID  Activity    SB ID    Creation Time          Procedure ID  State
----  ----------  -------  -------------------  --------------  ---------
   1  allocate    sbd-123  2023-01-06 13:56:47               1  COMPLETE
   2  observe     sbd-123  2023-01-06 13:56:56               2  PREPARED
```

Note that the first activity prepares and runs the script automatically but the second one only prepares the script but does not run it. To run the script of the second activity we need to note the `Procedure ID` for the activity and use `oet procedure` commands to run the script:

```
$ oet procedure start --pid=2

  ID   Script              Creation time        State
----  ---------------    -------------------  -------
  2   file://observe.py  2023-01-06 13:57:25  RUNNING
```

An `oet activity describe` command will give further detail on an activity.

```
$ oet activity describe --aid=1

   ID  Activity    SB ID      Procedure ID  State
 ----  ----------  -------  --------------  ---------
    1  allocate    sbd-123               1  COMPLETE

 URI                                       Prepare Only
 ----------------------------------------  --------------
 http://0.0.0.0:5000/api/v1.0/activities/1  False

 Time                        State
 --------------------------  ---------
 2023-01-06 13:56:47.655175  REQUESTED
 2023-01-06 13:56:47.934723  PREPARED
 2023-01-06 13:56:48.004753  RUNNING
 2023-01-06 13:56:50.382756  COMPLETE


 Script Arguments
 ----------------

 Method    Arguments    Keyword Arguments
 --------  -----------  -------------------
 init      [1, 'foo']   {'foo': 'bar'}
```

You can also view the details of the script that was run by the activity:

```
$ oet procedure describe --pid=1

   ID  Script               URI
 ----  ---------------      ----------------------------------------
    1  file://allocate.py   http://0.0.0.0:5000/api/v1.0/procedures/1

 Time                       State
 -------------------------  -------
 2023-01-06 13:56:47.655175  CREATING
 2023-01-06 13:56:47.663742  IDLE
 2023-01-06 13:56:47.665741  LOADING
 2023-01-06 13:56:47.730696  IDLE
 2023-01-06 13:56:47.731965  RUNNING 1
 2023-01-06 13:56:47.934723  READY
 2023-01-06 13:56:48.004753  RUNNING 2
 2023-01-06 13:56:50.382756  READY

Index       Method    Arguments     Keyword Arguments
--------   --------   ----------   -------------------
  1          init      [1, 'foo']    {'foo': 'bar'}
  2          run       []            {}
```

CHAPTER

FOUR

ENVIRONMENT VARIABLES

4.1 Telescope

The SKA comprises two telescopes: SKA MID (Dishes) and SKA LOW (Antennas). The behaviour of code in the ska_oso_scripting module differs depending on whether it is running in an SKA MID environment (default) or an SKA LOW environment. For example, when configured for SKA MID, the code will reject CDM payloads intended for SKA LOW.

The ska-oso-scripting code is configured for MID or LOW by setting the `SKA_TELESCOPE` environment variable to either 'skamid' or 'skalow'. If no environment variable is specified, the code assumes it is controlling SKA MID.

The telescope setting is also exposed as a configurable value in the ska-oso-scripting Helm charts, with a default value also set to SKA MID. The ska-oso-scripting definitions in the skamid and skalow SKAMPI Helm charts set the appropriate value for their respective deployments.

4.2 Tango Device FQDNs

The SKA, and so by extension the OET, makes use of Tango Controls to control the telescope hardware. The Fully Qualified Domain Names (FQDNs) or prefixes of the Tango devices used to control the central node (telescope) and sub-arrays are set as environment variables `CENTRALNODE_FQDN` and `SUBARRAYNODE_FQDN_PREFIX` respectively. These environment variables are set to the those defined in *values.yaml* when ska-oso-scripting/SKAMPI is deployed.

CHAPTER

FIVE

C&C VIEW: OET CLIENT AND OET BACKEND

This view is a component and connector (C&C) view of the OET that depicts the primary OET clients and their connection to the OET backend, and how the components of the backend are connected.

5.1 Primary Presentation

5.2 Elements and their properties

5.2.1 Components

Table 1: Key OET clients and core components of the OET backend

Component	Description
`FlaskWorker`	FlaskWorker is a Flask application that presents a RESTful OET API, functioning as a REST adapter for the ScriptExecutionService. Scripts can be created, controlled, and terminated via the REST API. The FlaskWorker presents a REST resource for each script process created and managed by the ProcessManager. FlaskWorker also presents a Server-Sent Event (SSE) data stream, republishing each event seen on the OET event bus as an SSE event. This SSE stream gives remote clients visibility of actions taken by the OET backend and events announced by scripting libraries and user scripts.
`main()`	The main component is the first component to be started when the OET backend is launched. It has two major responsibilities: first, it launches and thereafter manages the lifecycle of all components comprising the OET backend apart from the 'script process', whose lifecycle is managed separately by the script supervisor component. Second, it manages the OET event bus, routing OET events between backend components. The main component is responsible for establishing correct POSIX signal handling for the OET backend. For example, main instructs other OET backend components to terminate when a SIGHUP signal is received. The main component is the parent OS process of all other OET backend component processes.
OET Web UI	**NOT IMPLEMENTED YET** The OET Web UI is a web interface for the OET that can be used to submit SBs for execution. This interface is intended to operate from the perspective of SB execution rather than generic script execution, thus providing a more user-friendly interface than the OET CLI that an operator or tester could use until the OST is available.
OST	**NOT IMPLEMENTED YET** The SKA Online Scheduling Tool (OST) instructs the OET which SB should be executed next, taking into account aspects such as telescope resource availability, observing conditions, source visibility, science priority, etc.
`RestClientUI`	RestClientUI provides a command-line interface for invoke actions on the OET backend. The CLI is a general interface whose operations (currently) focus on the script execution perspective (load script, abort script, etc.) rather than the telescope-domain use cases (assign resources to subarray, execute SB, etc.). In addition to controlling script execution, the CLI can be used to inspect the status of scripts that have run or are running.
`ScriptExecut`	ScriptExecutionServiceWorker responds to requests received by the FlaskWorker, relaying the request to the ScriptExecutionService and publishing the response as an event that can be received by the FlaskWorker and returned to the user in the appropriate format.
ScriptExecut	ScriptExecutionService present the high-level API for script execution. The ScriptExecutionService orchestrates control of internal OET objects to satisfy an API request. ScriptExecutionService is also responsible for recording script execution history. ScriptExecutionService can return a presentation model of a script, its current state, and its execution history. See ProcedureSummary in the backend module view.
ScriptWorker	ScriptWorker represents the child Python process running the requested user script. For SKA operations, most scripts executed by the OET, and hence scripts that will run in a Script Process, will be 'observing scripts' that control an SKA subarray. The content and purpose of these 'observing scripts' is contained and defined in the ska-oso-scripting project. Note that the OET backend is independent of the content and function of the script, which could serve any purpose and is not limited to Tango-based telescope control.

5.2.2 Connectors

Table 2: Connectors between OET clients and the OET backend

Connectors	Description
REST over HTTP	REST over HTTP defines a request/response connector that is used by a client to invoke services on a server using REST over HTTP. Script processes are presented as REST resources by the OET backend. Using the REST over HTTP connector, clients can control the lifecycle and/or inspect the status of scripts running in the OET backend.
OET event bus	OET event bus connector defines an internal pub/sub connector used by an OET component to publish and subscribe to OET events (messages) sent on a topic.
Server-Sent Event	SSE connector defines a connector that is used by a client to listen to a continuous data stream of SSE events sent over a HTTP connection from an SSE server. SSE connectors have a client role and a server role. The SSE connector is used to give clients visibility of OET events published on the OET event bus.

5.3 Context

5.4 Variability Guide

The OET CLI reads the `OET_REST_URI` environment variable to find the URL of the OET REST server.

5.5 Rationale

5.5.1 REST over HTTP

REST over HTTP was selected as the protocol for remote control of the backend control for two reasons. First, we needed a protocol that was supported by multiple languages, anticipating that the OET web UI might not be Python based. Second, we preferred a stable and mature protocol with good library support. REST satisfies all these requirements, with good Python library support for both REST clients and REST servers.

5.5.2 Server-Sent Events

Insights into remote OET activities and script execution are obtained by monitoring events sent on the OET event bus. OET components, scripting libraries, and user scripts can all announce events of interest by publishing an event on the OET event bus. Events are published on various topics, from the script lifecycle (script loaded, script running, script aborting, script aborted, etc.), through to the SB lifecycle (SB resources allocated, observation started, observation complete, etc.) and subarray lifecycle (resources allocated, resources configured, scan started, etc.).

We needed a mechanism that would give the OET CLI, and possible the OET web UI at some future date, a tap into these events broadcast inside a remote OET backend. This use case requires the server to push events as they happen and have the client process/display them as they are received. Standard synchronous HTTP request/response does not map easily onto this use case and so we searched for a standard that would allow server-pushed messages. Any mechanism would also need to be language independent, mature, easily implemented and easily deployable in a Kubernetes setting, just as for REST over HTTP.

Server-Sent Events (SSE) was selected as it satisfies all of these criteria. SSE operates over HTTP, and the SSE API is standardised as part of HTML5. SSE has growing language support, including Python server and client library support, which helps keep the OET implementation simple. As it operates over HTTP, it can be delivered via the same Kubernetes ingress as the OET REST API.

5.5.3 No dedicated message broker

Systems that use a message-oriented architecture often use an dedicated message broker component such as RabbitMQ or Kafka whose sole responsibility is the delivery of messages to subscribers. Using a dedicated message broker can increase scalability by allowing multiple distributed brokers, increase reliability by allowing guaranteed message delivery, and promote system modifiability and composability by allowing routing of messages to inhomogeneous, loosely coupled, and potentially distributed subscribers via the network.

The OET does not currently use an external message broker as simplicity of deployment and reduced system complexity are currently prioritised over the advantages that an external message broker brings. Routing messages via a network broker would introduce complexity, overhead, and failure modes that are unnecessary in a homogeneous system with message publishers and message subscribers running in the same process space on the same host. We assume that message delivery through Python multiprocessing queues - essentially, communication via UNIX pipes - is robust and does not require message delivery guarantees. Additionally, telescope control scripts are not designed to be resumed in the event of failure, hence there is no value in resending any message lost to a failed ScriptWorker to a new replacement ScriptWorker. There is also a desire to keep the OET deployment footprint small and with minimal dependencies so that the OET can be easily incorporated and/or deployed in a simulator context for other OSO use.

That said, the OET architecture does allow the introduction of a dedicated message broker if the OET requirements change.

CHAPTER

SIX

MODULE VIEW: SCRIPT EXECUTION UI AND SERVICE API

Note: Diagrams are embedded as SVG images. If the text is too small, please use your web browser to zoom in to the images, which should be magnified without losing detail. Alternatively open image in a new tab with right click + Open in a new tab.

This view is a module view showing the key components responsible for the OET interface, how they relay requests from remote OET clients to the internal OET components responsible for meeting that request, and how the response makes its way back to the client.

6.1 Primary Presentation

Fig. 1: Major classes involved in the user interface and remote control of the script execution API.

6.2 Element Catalogue

6.2.1 Elements and Their Properties

Component	Description
app (variable in `startup`)	app is the Flask web application that makes the OET available over HTTP. app is the local variable created during FlaskWorker startup. The web application has the API blueprint and ServerSentEventsBlueprint registered, which makes the OET REST API and the OET event stream available when the web app is run.
`ProcedureAPI`	ProcedureAPI is a Flask blueprint containing the Python functions that implement the OET REST API. HTTP resources in this blueprint are accessed and modified to control script execution. As the resources are accessed, the API implementation publishes an equivalent request event, which triggers the ScriptExecutionServiceWorker to take the appropriate action to satisfy that request. API also converts the response back to a suitable HTML response. The REST API is documented separately in *Module View: REST API*.
Blueprint	A Flask Blueprint collects a set of HTTP operations that can be registered on a Flask web application. Registering a Blueprint to a Flask application makes the HTTP operations in that blueprint available when the web application is deployed.
`EventBusWorke`	EventBusWorker is a base class that bridges the independent pypubsub publish-subscribe networks so that a pypubsub message seen in one EventBusWorker process is also seen by other EventBusWorker processes. EventBusWorker is intended to be inherited by classes that register their methods as subscribers to pypubsub topics, so that the subclass method is called whenever an event on that topic is received.
Flask	Flask (https://flask.palletsprojects.com) is a third-party Python framework for developing web applications. It provides an easy way to expose a Python function as a HTTP endpoint. Flask is used to present the functions in the restserver module as HTTP REST resources.
`FlaskWorker`	FlaskWorker runs the 'app' Flask application. As a subclass of EventBusWorker, FlaskWorker also relays pypubsub messages to and from other Python processes.
`mptools`	mptools is a Python framework for creating robust Python applications that run code concurrently in independent Python processes. See *Module view: Script Execution* for details.
`PrepareProce`	PrepareProcessCommand encapsulates all the information required to prepare a script for execution. It references both the script location and arguments that should be passed to the script initialisation function, if such a function is present.
`ProcedureHis`	ProcedureHistory represents the state history of a script execution process, holding a timeline of state transitions and any stacktrace resulting from script execution failure.
`ProcedureSum`	ProcedureSummary is a presentation model capturing information on a script and its execution history. Through the ProcedureSummary, information identifying the script, the process running it, the current and historic process state, plus a timeline of all function called on the script and any resulting stacktrace can be resolved.
pypubsub	pypubsub (https://pypubsub.readthedocs.io) is a third-party Python library that provides an implementation of the Observer pattern. It provides a publish-subscribe API for that clients can use to subscribe to topics. pypubsub notifies each subscriber whenever a message is received on that topic, passing the message to the client. pypubsub offer in-process publish-subscribe; it has no means of communicating messages to other Python processes.
`RestClientUI`	RestClientUI is a command line utility that accesses the OET REST API over the network. The RestClientUI provides commands for creating new script execution processes, invoking methods on user scripts, terminating scrip execution, listing user processes on the remote machine, and inspecting the state of a particular user script process.
`ScriptExecut`	ScriptExecutionService provides the high-level API for the script execution domain, presenting methods that 'start script X' or 'run method Y of user script Z'. See *Module view: Script Execution* for details on how this is achieved. In addition to its primary responsibility of triggering actions in response to API calls, ScriptExecutionService is also responsible for recording script execution history, which it achieves by monitoring for and recording script lifecycle change events. ScriptExecutionService manages the history state so that the number of records does not increase in an unbounded manner (currently, history is maintained for all active scripts and a maximum of 10 inactive scripts (=any script that is complete). ScriptExecutionService provides a presentation model of a script and its execution history, which can be formatted for presentation via the REST service and CLI. This presentation model

6.2.2 Element Interfaces

The major interface between the UI and OET backend is the REST API presented by the FlaskWorker, which is documented separately in *Module View: REST API*.

6.2.3 Element Behaviour

API invocation via HTTP REST

The sequence diagram below illustrates how the components above interact to invoke a call on an remote ScriptExecutionService instance in response to a request from a client. This diagram shows how the user request is received by the FlaskWorker REST backend, how that triggers actions on independent ScriptExecutionServiceWorker process hosting the ScriptExecutionService instance, and how the response is returned to the user.

Inter-process publish-subscribe

The sequence diagram below illustrates how in-process pypubsub messages are communicated to other processes, which is an essential part of the communication between FlaskWorker and ScriptExecutionServiceWorker and forms the basis for how event messages emitted by scripts can be published to the outside world in an HTTP SSE stream.

6.3 Context Diagram

6.4 Variability Guide

N/A

6.5 Rationale

N/A

CHAPTER

SEVEN

MODULE VIEW: ACTIVITY UI AND SERVICE API

Note: Diagrams are embedded as SVG images. If the text is too small, please use your web browser to zoom in to the images, which should be magnified without losing detail. Alternatively open image in a new tab with right click + Open in a new tab.

This view is a module view depicting the key components involved in SB activity execution; that is, requesting an activity described by a Scheduling Block to be run.

7.1 Primary Presentation

Fig. 1: Major classes responsible for the execution and management of activities.

7.2 Element Catalogue

7.2.1 Elements and their properties

Component	Description
ActivityStat	ActivityState is an enumeration defining the states that an Activity (a concept linking Scheduling Blocks to Procedures) can be in. State machine for activities has not yet been completely defined and currently Activity can only be in state `REQUESTED`.
ActivityServ	ActivityService provides the high-level API for the activity domain, presenting methods that 'run a script referenced by activity *X* of scheduling block *Y*'. The ActivityService completes user requests by translating the activity requests into Procedure domain commands which then execute the scripts. The steps taken by the ActivityService to construct a PrepareProcedureCommand are: 1. retrieve the Scheduling Block by ID from the ODA using the ODA client library 2. write Scheduling Block to a JSON file as */tmp/sbs/<sb_id>-<version>-<timestamp>.json* 3. convert PDM FileSystemScript object referenced by the given SB and activity to OET FileSystemScript 4. create a collection of init and run arguments by combining user-defined functions arguments with arguments listed in the SB 5. add the path to previously written SB JSON file to the run function arguments under key *sb_json* 6. create PrepareProcessCommand using the FileSystemScript object and arguments for *init* function from the collection of function arguments After the prepare command has been sent, it will wait for a response to record the procedure ID of the script relating to the activity. If *prepare_only* argument is set to false, ActivityService will create a StartProcessCommand and send it to the ScriptExecutionServiceWorker. It will include the Procedure ID, and request that function named *run* will be executed with the corresponding arguments.
`ActivityServ`	For a the OET REST deployment, ActivityServiceWorker is the client sending requests to the ActivityService. ActivityWorker responds to requests received by the FlaskWorker, relaying the request to the *ActivityService* and publishing the response as an event that can be received by the FlaskWorker and returned to the user in the appropriate format.

7.2.2 Element Interfaces

The major public interface in these interactions is the ActivityService API. For more information on this interface, please see the *Module View: REST API*.

7.2.3 Element Behaviour

Activity API invocation via HTTP REST

The sequence diagram below illustrates how the components above interact to invoke a call on an remote ActivityService instance in response to a request from a client. This diagram shows how the user request is received by the FlaskWorker REST backend, how that triggers actions on independent ActivityWorker process hosting the ActivityService instance, and how the response is returned to the user

Inter-process publish-subscribe

The Activity domain uses the same publish-subscribe system as Procedure domain for both communication between FlaskWorker and ActivityServiceWorker and for the ActivityService to communicate with the ScriptExecutionServiceWorker. For a diagram illustrating the flow of in-process pypubsub messages, see the *Inter-process publish-subscribe section* in the script execution API documentation.

7.3 Variability Guide

N/A

7.4 Rationale

7.4.1 Storing Scheduling Block in the Filesystem

It is currently only possible to deploy the activity and procedure services as one service. This means that the Scheduling Block can be written to file by the ActivityService and it will still be available to the Procedure domain. In the future the Activity and Procedure related services could be deployed in different locations so the current approach of saving SBs to a file should be refactored so that the script running on a different server can also access the SB.

7.4.2 Scheduling Block URI

Currently the Scheduling Block URI used in the OET system is a simple path string to a JSON file referred to by a keyword argument *sb_json*. In the future this will be expanded into a proper URI with several allowed prefixes such as *file://* for SB located in a file and *oda://* for SB that should be retrieved from the ODA.

CHAPTER

EIGHT

MODULE VIEW: SCRIPT EXECUTION

Note: Diagrams are embedded as SVG images. If the text is too small, please use your web browser to zoom in to the images, which should be magnified without losing detail. Alternatively open image in a new tab with right click + Open in a new tab.

This view is a module view depicting the key components involved in script execution; that is, creating new Python processes that load a user script and run functions in that user script when requested.

8.1 Primary Presentation

Fig. 1: Major classes responsible for the execution and management of user scripts.

8.2 Element Catalogue

8.2.1 Elements and their properties

Component	Description
Embedded-StringScript	*NOT IMPLEMENTED YET* EmbeddedStringScript holds a complete Python script as a string. This class has been identified as possibly being useful as it allows a SchedulingBlock to directly specify and inject the code to be run, but has not been implemented.
Environment	Environment is a dataclass that holds the information required to identify a Python virtual environment and its location on disk. In addition, it holds synchronisation primitives to avoid race conditions between multiple requests to create the same environment, as would be the case for multiple requests to create virtual environments for the same git project and git commit hash.
EnvironmentM	EnvironmentManager is responsible for creating and managing Environments, the custom Python virtual environments in which a user script that requiring a non-default environment runs. Typically, this is the case for a request to run a script located in a git repository, where the request requires a more recent version of the ska-oso-scripting library or control scripts than was packaged with the OET. Environment creation can be expensive, typically taking 20-30 seconds to ready a new ska-oso-scripting environment and with all-new dependencies. For this reason, EnvironmentManager is designed to allow virtual environments to be shared for script execution requests that target the same git repository and commit, as uniquely identified by the git commit hash. EnvironmentManager currently has no policy for deleting virtual environments, and the number of virtual environments could in principle increase unbounded manner. A policy of maintaining all active environments and maintaining a maximum of *n* inactive environments is expected to be implemented.
Event	The Event class manages a flag that can be set and/or inspected by multi Python processes. Events are commonly used to signify to observers of the Event that a condition has occurred. Event is part of the standard Python library.
EventBusWork	EventBusWorker is a QueueProcWorker that relays pubsub events seen in one EventBusWorker process to other EventBusWorker processes. See *Module view: Script Execution UI and Service API* for more information.
ExecutableSc	ExecutableScript is an abstract class for any class that defines a Python script to be executed.
FilesystemSc	FilesystemScript captures the information required to run a Python script located within the filesystem of a deployed OET backend. As an example, in a Kubernetes context this could point to a script contained in the default preinstalled scripting environment, or a script made available in a persistent volume mounted by the OET pod.
GitScript	GitScript captures the information required to run a Python script that is located in a git repository. It collects a set of identifying information that together can conclusively identify the specific script to be run, such as git repository, branch, tag, and commit hash.
MainContext	MainContext is the parent context for a set of worker processes that communicate via message queues. It defines a consistent architecture for event-based communication between Python processes and consistent behaviour for POSIX signal handling and process management. MainContext is responsible for routing messages between the ProcWorkers created within the scope of a MainContext. MainContext is also responsible for managing the termination of the child processes, first requesting that the child process co-operate and stop execution cleanly, before escalating and using increasingly forceful means to terminate unresponsive processes (e.g., SIGINT, then SIGHUP). Lastly, MainContext is responsible for the correct management of the Python multiprocessing primitives created within the scope of the MainContext that are used for inter-process communication and synchronisation.
MPQueue	MPQueue is an extension of the standard library multiprocessing.Queue that adds get/set methods that return booleans when the operation fails rather than raising exceptions, which makes the class easier to use in some contexts.
Proc	Proc represents a child Python process of a MainContext. Proc instances exist in the scope of a MainContext instance and in the same OS process as the parent MainContext. Procs are the MainContext's link to the ProcWorkers running in an independent operating system process with an independent Python interpreter. Every ProcWorker running in a child process is associated with one Proc. Each Proc is responsible for bootstrapping its ProcWorker and managing its lifecycle. Proc ar-

8.2.2 Element Interfaces

The major public interface in these interactions is the ScriptExecutionService API. For more information on this interface, please see the API documentation for *ScriptExecutionService*.

8.2.3 Element Behaviour

ScriptExecutionService

The sequence diagram below gives a high-level overview of how the *ScriptExecutionService* controls objects in the domain module to meet requests to prepare, start, and stop user script execution.

ScriptExecutionService.prepare

The diagram below gives more detail on how the domain layer handles a request to prepare a script for execution.

ScriptWorker

The diagram below illustrates how a *ScriptWorker* is created and how it communicates startup success with the parent process.

ScriptWorker.main_loop

The diagram below depicts the main *ScriptWorker* message loop, illustrating how the various messages from the parent *ProcessManager* are handled by child *ScriptWorker*.

8.3 Context Diagram

8.4 Variability Guide

N/A

8.5 Rationale

N/A

CHAPTER

NINE

MODULE VIEW: REST API

9.1 1. Interface Identity

OET REST API presents REST resources that can be used to manage the lifecycle of Python scripts running on a remote server and to inspect their status.

9.2 2. Resources

A 'Procedure' represents a Python script to run, or that is running, on the backend. The REST API operates on Procedures.

The standard workflow is to use the API to:

1. Instruct the backend to prepare a script for execution by using HTTP POST to upload a JSON `Procedure` to /api/v1/procedures
2. Start script execution by uploading an updated JSON `Procedure` with a `ProcedureState` of `RUNNING`.
3. (optional) a running script can be terminated by using PUT to upload a JSON `Procedure` with a `ProcedureState` of `STOPPED`.

The current status of a script execution can be inspected at any time by reading the JSON `Procedure` with HTTP GET.

This workflow has been mapped to the following REST resources:

Table 1: Procedure REST resources

HTT Met	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 realtime OET events** Return an SSE data stream of OET events as they are published by the OET and scripts.
POS	`/api/v1/procedures`	**Prepare a new procedure** Loads the requested script and prepares it for execution.
PUT	`/api/v1/procedures/<id>`	**Modify the state of a prepared procedure** This can be used to start execution by setting the Procedure `state` attribute to `RUNNING` or stop execution by setting `state` to `STOPPED`.

An 'Activity' represents an action which a user will command the telescope to perform, eg 'allocate'

Table 2: Activity REST resources

HTT Met	Resource URL	Description
GET	/api/v1/ activities	**List activities** Return the collection of all activities.
GET	/api/v1/ activities/ <activity_id>	**Get activity** Return the a summary of the activity with given id.
POS	/api/v1/ activities	**Prepare a new activity** Loads the script from the SBDefinition for the given activity and prepares it for execution. Response is an ActivitySummary

9.3 3. Data Types and Constants

9.3.1 Type: `Procedure`

`Procedure` is used to represent a Python script running in a Python process on the OET backend. Attributes are:

- `string uri`: read-only URI of this procedure. Defined by the server on procedure creation.
- `FileSystemScript/GitScript script`: Script details containing script_uri, e.g., `file:///path/to/obsscript.py`, and additional information like git arguments.
- `CallArgs script_args`: arguments provided to the script at initialisation time and main execution time.
- `ProcedureState state`: current state of this `Procedure`.
- `ProcedureHistory history`: timestamped execution history for this `Procedure`.

Example

Below is an example `Procedure` JSON object. 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 `READY`:

```
{
    "script_args": {
      "init": {
        "args": [],
        "kwargs": {
          "sb_uri": "file:///path/to/scheduling_block_123.json",
          "subarray_id": 1
        }
      },
      "run": {
        "args": [],
        "kwargs": {}
      }
```

(continues on next page)

(continued from previous page)

```
    },
    "script": {
        "script_type": "filesystem",
        "script_uri": "file:///path/to/observing_script.py",
    },
    "history": {
        "process_states": [
           ("CREATING", 1601463545.57689632),
           ("IDLE", 1601463545.57843814),
           ("LOADING", 1601463545.58043561),
           ("IDLE", 1601463545.58865546),
           ("RUNNING", 1601463545.62904726),
           ("READY", 1601463545.7789776)
        ],
        "stacktrace": null
    },
    "state": "READY",
    "uri": "http://localhost:5000/api/v1.0/procedures/1"
}
```

If user wanted to run script located in a git repository `http://gitrepo.git` in branch `test`, the script JSON would look as below:

```
{ ...
    "script": {
        "script_type": "git",
        "script_uri": "git:///path/to/observing_script.py",
        "git_args": {"git_repo": "http://gitrepo.git", "git_branch": "test"}
    } ...
}
```

9.3.2 Type: `FileSystemScript`

`FileSystemScript` represents the script to be run from the file system. It has `script_uri` argument which points to an observing script present on the file system and `script_type` which has the value of `filesystem`.

9.3.3 Type: `GitScript`

`GitScript` inherits from `FileSystemScript`, which means it also has a `script_uri` argument and `script_type` of `git`. Additionally it has an argument, `GitArgs` which points to the git repository the given script is located in. The arguments for `GitArgs` are:

- `git_repo` which points to the full URL of the repository
- `git_branch` if specifying other than the default `master` branch
- `git_commit` if wanting to point to a specific commit within the repository.

9.3.4 Type: CallArgs

CallArgs represents the arguments to be passed to functions in the user script when those functions are called. Attributes are:

- FunctionArgs init: arguments passed to the script init function at script creation and initialisation time.
- FunctionArgs run: arguments passed to the script main function when the main execution function is called.

9.3.5 Type: FunctionArgs

FunctionArgs captures the positional arguments and keywords arguments (to be) passed to a Python function. Attribute are:

- list args: list of positional arguments for the Python function, e.g., "args": [1, 2, 3]
- dict kwargs: dictionary of keywords arguments, e.g., "kwargs": {"subarray_id": 1}

9.3.6 Type: ProcedureState

ProcedureState is an enumeration representing the current lifecycle state of the Python process running the user script. It can be one of:

- IDLE: state between script preparation steps where no action is ongoing.
- CREATING: script creation has been started.
- LOADING: loading the specified script file to be executed.
- READY: script is ready to run specified function, e.g. init or main.
- RUNNING: script is running, i.e., the script's init or main function is currently executing.
- STOPPED: script was terminated by the OET before the script could complete.
- COMPLETE: the script completed successfully, i.e., the main function completed and no exception was raised.
- FAILED: an exception was raised during script preparation or execution.

9.3.7 Type: ProcedureHistory

ProcedureHistory represents a timeline of ProcedureStates that the Procedure has passed through. Attributes are:

- list process_states: a List of ProcedureStates and timestamps when that ProcedureState was reached, e.g. process_states: [('CREATING', 18392174.543), ('RUNNING', 18392143.546), ('COMPLETE', 183925456.744)].
- string stacktrace: populated with the stacktrace from the script if the final ProcedureState is FAILED. This attribute is set to None for any other final state.

9.4 4. Error Handling

Accessing the URL of a `Procedure` that does not exist on the backend or whose history has expired will result in a HTTP 404 error:

```
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"}
```

9.5 5. Variability

None

9.6 6. Quality Attribute Characteristics

None

9.7 7. Rationale and Design Issues

The procedure history is limited, and at some point a Procedure REST resource will become unavailable as it becomes superseded by new Procedures and that history slot is reclaimed. This is not expected to be a problem as a maximum of one script can run at any one time, so even a small history allows a reasonable amount of time for that Procedure history to be inspected.

9.8 8. Usage Guide

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.

9.8.1 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
```

(continues on next page)

(continued from previous page)

```
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": {
        "args": [],
        "kwargs": {
          "sb_uri": "file:///path/to/scheduling_block_123.json",
          "subarray_id": 1
        }
      },
      "run": {
        "args": [],
        "kwargs": {}
      }
    },
    "script": {
        "script_type": "filesystem",
        "script_uri": "file:///path/to/observing_script.py"
    },
    "history": {
        "process_states": [
           ("CREATING", 1601463545.7589678),
           ("IDLE", 1601463545.7598525),
           ("LOADING", 1601463545.7649524),
           ("IDLE", 1601463545.7668241),
           ("RUNNING", 1601463545.7694371),
           ("READY", 1601463545.7748005)
        ],
        "stacktrace": null
    },
    "state": "READY",
    "uri": "http://localhost:5000/api/v1.0/procedures/2"
  }
}
```

9.8.2 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
```

(continues on next page)

(continued from previous page)

```
Date: Wed, 15 Jan 2020 10:11:42 GMT

{
  "procedures": [
    {
      "script_args": {
        "init": {
          "args": [],
          "kwargs": {
            "dishes": [
              1,
              2,
              3
            ]
          }
        },
        "run": {
          "args": [],
          "kwargs": {}
        }
      },
      "script": {
        "script_type": "filesystem",
        "script_uri": "file:///path/to/resource_allocation.py"
      },
      "history": {
          "process_states": [
               ("CREATING", 1601463545.7589678),
               ("IDLE", 1601463545.7598525),
               ("LOADING", 1601463545.7649524),
               ("IDLE", 1601463545.7668241),
               ("RUNNING", 1601463545.7694371),
               ("READY", 1601463545.7748005)
            ],
          "stacktrace": null
              },
      "state": "READY",
      "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": {}
        }
```

(continues on next page)

(continued from previous page)

```
      },
      "script": {
        "script_type": "filesystem",
        "script_uri": "file:///path/to/observing_script.py"
      },
      "history": {
           "process_states": [
               ("CREATING", 1601463545.7589678),
               ("IDLE", 1601463545.7598525),
               ("LOADING", 1601463545.7649524),
               ("IDLE", 1601463545.7668241),
               ("RUNNING", 1601463545.7694371),
               ("READY", 1601463545.7748005)
           ],
           "stacktrace": null
      },
      "state": "READY",
      "uri": "http://localhost:5000/api/v1.0/procedures/2"
    }
  ]
}
```

9.8.3 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

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

(continues on next page)

(continued from previous page)

```
    "script": {
      "script_type": "filesystem",
      "script_uri": "file:///path/to/resource_allocation.py"
    },
    "history": {
        "process_states": [
           ("CREATING", 1601463545.7589678),
           ("IDLE", 1601463545.7598525),
           ("LOADING", 1601463545.7649524),
           ("IDLE", 1601463545.7668241),
           ("RUNNING", 1601463545.7694371),
           ("READY", 1601463545.7748005)
        ],
        "stacktrace": null
    },
    "state": "READY",
    "uri": "http://localhost:5000/api/v1.0/procedures/1"
  }
}
```

9.8.4 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": {
        "args": [],
        "kwargs": {
          "sb_uri": "file:///path/to/scheduling_block_123.json",
          "subarray_id": 1
        }
      },
      "run": {
        "args": [],
        "kwargs": {
          "scan_duration": 14.0
```

(continues on next page)

(continued from previous page)

```
        }
      }
    },
    "script": {
      "script_type": "filesystem",
      "script_uri": "file:///path/to/observing_script.py"
    },
    "history": {
        "process_states": [
           ("CREATING", 1601463545.7589678),
           ("IDLE", 1601463545.7598525),
           ("LOADING", 1601463545.7649524),
           ("IDLE", 1601463545.7668241),
           ("RUNNING", 1601463545.7694371),
           ("READY", 1601463545.7748005)
         ],
        "stacktrace": null
    }
    "state": "READY",
    "uri": "http://localhost:5000/api/v1.0/procedures/2"
  }
}
```

9.8.5 Terminate 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 "}
```

9.8.6 Listen to 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
```

(continues on next page)

(continued from previous page)

```
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

TEN

SKA_OSO_OET.TANGO

The ska_oso_oet.tango module contains code that could be called from observing scripts. Primarily, this will involve interactions with *ska_oso_oet.tango.TangoExecutor*.

class ska_oso_oet.tango.**TangoExecutor**(*proxy_factory=<ska_oso_oet.tango.TangoDeviceProxyFactory object>*)

TangoExecutor is the proxy between calling code and Tango devices. It accepts encapsulated Tango interactions and performs them on behalf of the calling code.

__init__(*proxy_factory=<ska_oso_oet.tango.TangoDeviceProxyFactory object>*)

Create a new TangoExecutor.

Parameters
proxy_factory – a function or object which, when called, returns an object that conforms to the PyTango DeviceProxy interface.

execute(*command:* Command, ***kwargs*)

Execute a Command on a Tango device.

Additional kwargs to the DeviceProxy can be specified if required.

Parameters
command – the command to execute

Returns
the response, if any, returned by the Tango device

read(*attribute:* Attribute)

Read an attribute on a Tango device.

Parameters
attribute – the attribute to read

Returns
the attribute value

read_event(*attr:* Attribute) → tango.EventData

Get an event for the specified attribute.

subscribe_event(*attribute:* Attribute)

Subscribe event on a Tango device.

Parameters
attribute – the attribute to subscribe to

Returns
subscription ID

unsubscribe_event(*attribute:* Attribute, *event_id: int*)

unsubscribe event on a Tango device.

Parameters

- **attribute** – the attribute to unsubscribe
- **event_id** – event subscribe id

Returns

class ska_oso_oet.tango.**Attribute**(*device: str*, *name: str*)

An abstraction of a Tango attribute.

__init__(*device: str*, *name: str*)

Create an Attribute instance.

Parameters

- **device** – the FQDN of the target Tango device
- **name** – the name of the attribute to read

class ska_oso_oet.tango.**Command**(*device: str*, *command_name: str*, **args*, ***kwargs*)

An abstraction of a Tango command.

__init__(*device: str*, *command_name: str*, **args*, ***kwargs*)

Create a Tango command. :param device: the FQDN of the target Tango device :param command_name: the name of the command to execute :param args: unnamed arguments to be passed to the command :param kwargs: keyword arguments to be passed to the command

CHAPTER

ELEVEN

SKA_OSO_OET.FEATURES

The features module contains code handling the setting and reading of OET feature flags. OET feature flags are configured once, at deployment time, and are not reconfigured during execution.

Feature flag values are set from, in order:

1. environment variables,
2. an .ini file
3. default values set in code

class ska_oso_oet.features.**Features**(*config_parser: ConfigParser*)

The Features class holds flags for OET features that can be toggled.

__init__(*config_parser: ConfigParser*)

static create_from_config_files(**paths*) → *Features*

Create a new Features instance from a set of feature flag configuration files.

Parameters

paths – configuration files to parse

CHAPTER

TWELVE

SKA_OSO_OET

Reading ska_oso_oet.ini file value and initializing constant of feature toggle with enabling event based polling/pubsub

12.1 ska_oso_oet.main

12.2 ska_oso_oet.tango

class ska_oso_oet.tango.**TangoDeviceProxyFactory**

A call to create Tango DeviceProxy clients. This class exists to allow unit tests to override the factory with an implementation that returns mock DeviceProxy instances.

class ska_oso_oet.tango.**TangoExecutor**(*proxy_factory=<ska_oso_oet.tango.TangoDeviceProxyFactory object>*)

TangoExecutor is the proxy between calling code and Tango devices. It accepts encapsulated Tango interactions and performs them on behalf of the calling code.

class SingleQueueEventStrategy(*mgr:* SubscriptionManager)

SingleQueueEventStrategy encapsulates the event handling behaviour of the TangoExecutor from ~October 2021, when all events were added to a single queue and subscriptions were created and released after each attribute read operation.

We hope to replace this with a more advanced implementation that allows subscriptions to multiple events.

Parameters
mgr – SubscriptionManager instance used to observe events

__init__(*mgr:* SubscriptionManager)

notify(*evt: tango.EventData*)

This implements the SubscriptionManager EventObserver interface. Tango ChangeEvents republished by the SubscriptionManager are received via this method.

Queue is thread-safe so we do not need to synchronise this method with read_event.

read_event(*attr:* Attribute) → tango.EventData

Read an event from the queue. This function blocks until an event is received.

With a single subscription active at any one time, the attribute is ignored by this implementation but is expected to be required by strategy that support multiple attribute subscriptions.

subscribe_event(*attr:* Attribute) → int

Subscribe to change events published by a Tango attribute.

This strategy only supports one active subscription at any time. An exception will be raised if a second subscription is attempted.

This method returns a subscription identifier which should be supplied to a subsequent unsubscribe_event method.

Parameters
attr – attribute to subscribe to

Returns
subscription identifier

unsubscribe_event(*attr:* Attribute, *subscription_id:* *int*) → None

Unsubscribe to change events published by a Tango attribute.

This strategy only supports one active subscription at any time. An exception will be raised if a second subscription is attempted.

Parameters
- **attr** – attribute to unsubscribe from
- **subscription_id** – subscription identifier

__init__(*proxy_factory=<ska_oso_oet.tango.TangoDeviceProxyFactory object>*)

Create a new TangoExecutor.

Parameters
proxy_factory – a function or object which, when called, returns an object that conforms to the PyTango DeviceProxy interface.

execute(*command:* Command, ***kwargs*)

Execute a Command on a Tango device.

Additional kwargs to the DeviceProxy can be specified if required.

Parameters
command – the command to execute

Returns
the response, if any, returned by the Tango device

read(*attribute:* Attribute)

Read an attribute on a Tango device.

Parameters
attribute – the attribute to read

Returns
the attribute value

read_event(*attr:* Attribute) → tango.EventData

Get an event for the specified attribute.

subscribe_event(*attribute:* Attribute)

Subscribe event on a Tango device.

Parameters
attribute – the attribute to subscribe to

Returns
subscription ID

unsubscribe_event(*attribute:* Attribute, *event_id: int*)

unsubscribe event on a Tango device.

Parameters

- **attribute** – the attribute to unsubscribe
- **event_id** – event subscribe id

Returns

class ska_oso_oet.tango.**SubscriptionManager**(*proxy_factory=<ska_oso_oet.tango.TangoDeviceProxyFactory object>*)

SubscriptionManager is a proxy for Tango event subscriptions that prevents duplicate subscriptions and minimises subscribe/unsubscribe calls.

Previously, each time a script listened to an event, it would subscribe to an event, wait for reception of the appropriate event, then unsubscribe. These multiple subscribe/unsubscribe calls were found to create problems. SubscriptionManager was introduced to manage subscriptions, with the aim of having fewer, longer-lived subscriptions. Clients subscribe to the SubscriptionManager, and the SubscriptionManager handles any required subscriptions to Tango devices.

The SubscriptionManager component is responsible for managing events and event subscriptions in the OET. The SubscriptionManager sits as a proxy between client and Tango event subscriptions, moving the pub/sub layer accessed by clients away from the Tango layer and into the OET layer. Clients register with the SubscriptionManager as observers of an attribute. If required, one long-lived Tango subscription per attribute is created on demand by the SubscriptionManager. The SubscriptionManager relays received Tango events to all attribute observers registered at the time of event reception. Unregistering an observer from the SubscriptionManager prevents subsequent notifications but does not affect the underlying Tango event subscription, which continues to operate until the Python interpreter exits.

Legacy calling code expects a maximum of one subscription to be active at any one time. Additionally, the caller always sandwiched `read_event` calls between `subscribe_attribute` and `unsubscribe_attribute` calls. Together, this meant subscriptions were short-lived, existing for the duration of a single attribute monitoring operation, and that one Queue to hold events was sufficient as there would only ever be one Tango event subscription. To maintain this legacy behaviour, `subscribe_attribute` and `unsubscribe_attribute` register and unregister the TangoExecutor as an observer of events, with the `TangoExecutor.notify` method adding received events to the TangoExecutor queue read by the legacy `TangoExecutor.read_event` method.

Fig. 1: Class diagram for components involved in OET event handling

Fig. 2: Sequence diagram from OET event handling

Members

__init__(*proxy_factory=<ska_oso_oet.tango.TangoDeviceProxyFactory object>*)

register_observer(*attr:* Attribute, *observer*)

Register an EventObserver as an observer of a Tango attribute.

Once registered, the EventObserver will be notified of each Tango event published by the attribute.

Parameters

- **attr** – Tango attribute to observe
- **observer** – the EventObserver to notify

unregister_observer(*attr:* Attribute, *observer*)

Deregister an EventObserver as an observer of a Tango attribute.

Parameters

- **attr** – the observed Tango attribute
- **observer** – the EventObserver to unsubscribe

class ska_oso_oet.tango.**LocalScanIdGenerator**(*start=1*)

LocalScanIdGenerator is an abstraction of a service that will generate scan IDs as unique integers. Expect scan UID generation to be a database operation or similar in the production implementation.

__init__(*start=1*)

next()

Get the next scan ID.

Returns
integer scan ID

property value

Get the current scan ID.

class ska_oso_oet.tango.**RemoteScanIdGenerator**(*hostname*)

RemoteScanIdGenerator connects to the skuid service to retrieve IDs

__init__(*hostname*)

next()

Get the next scan ID.

Returns
integer scan ID

property value

Get the current scan ID.

class ska_oso_oet.tango.**Callback**

Callback is an observable that distributes Tango events received by the callback instance to all observers registered at the moment of event reception.

__init__()

notify_observers(*evt: tango.EventData*)

Distribute an event to all registered observers.

Parameters
evt – event to distribute

register_observer(*observer*)

Register an EventObserver.

Once registered, the observer will be notified of all Tango events received by this instance.

Parameters
observer – observer to register

unregister_observer(*observer*)

Unregister an EventObserver.

Unsubscribed observers will not receive Tango events subsequently received by this instance.

Parameters

observer – observer to register

12.3 ska_oso_oet.ui

CHAPTER

THIRTEEN

SKA_OSO_OET.ACTIVITY

13.1 ska_oso_oet.activity.application

The ska_oso_oet.activity.application module contains code related to OET 'activities' that belong in the application layer. This application layer holds the application interface, delegating to objects in the domain layer for business rules and actions.

class ska_oso_oet.activity.application.**ActivityService**

ActivityService provides the high-level interface and facade for the activity domain.

The interface is used to run activities referenced by Scheduling Blocks. Each activity will run a script (or *procedure*) but ActivityService will create the necessary commands for Procedure domain to create and execute the scripts.

__init__()

complete_run_activity(*prepared_summary:* ProcedureSummary, *request_id: int*) → *ActivitySummary* | None

Complete the request to run the Activity, using the ProcedureSummary that is now available. This includes updating the Activity with the procedure_id, sending the request to start the procedure if prepare_only is not set to True, and returning the ActivitySummary.

Parameters

- **prepared_summary** – the ProcedureSummary for the Procedure related to the requested Activity
- **request_id** – The original request_id from the REST layer

Returns

an ActivitySummary describing the state of the Activity that the Procedure is linked to, or None if the Procedure was not created from an Activity

prepare_run_activity(*cmd: ActivityCommand*, *request_id: int*) → None

Prepare to run the activity of a Scheduling Block. This includes retrieving the script from the scheduling block and sending the request messages to the ScriptExecutionService to prepare the script.

The request_id is required to be propagated through the messages sent to the Procedure layer, so the REST layer can wait for the correct response event.

Parameters

- **cmd** – dataclass argument capturing the activity name and SB ID
- **request_id** – The original request_id from the REST layer

summarise(*activity_ids: List[int] | None = None*) → List[*ActivitySummary*]

Return ActivitySummary objects for Activities with the requested IDs.

This method accepts an optional list of integers, representing the Activity IDs to summarise. If the IDs are left undefined, ActivitySummary objects for all current Activities will be returned.

Parameters
activity_ids – optional list of Activity IDs to summarise.

Returns
list of ActivitySummary objects

write_sbd_to_file(*sbd*) → str

Writes the SBD json to a temporary file location and returns the path.

class ska_oso_oet.activity.application.**ActivitySummary**(*id: int*, *pid: int*, *sbd_id: str*, *activity_name: str*, *prepare_only: bool*, *script_args: Dict[str, ska_oso_oet.procedure.domain.ProcedureInput]*, *activity_states: List[Tuple[ska_oso_oet.activity.domain.ActivityState, float]]*)

__init__(*id: int*, *pid: int*, *sbd_id: str*, *activity_name: str*, *prepare_only: bool*, *script_args: Dict[str, ProcedureInput]*, *activity_states: List[Tuple[ActivityState, float]]*) → None

13.2 ska_oso_oet.activity.domain

The ska_oso.activity.domain module contains code that belongs to the activity domain layer. Classes and definitions contained in this domain layer define the high-level concepts used to describe and launch scheduling block activities.

class ska_oso_oet.activity.domain.**Activity**(*activity_id: int*, *procedure_id: int | None*, *sbd_id: str*, *activity_name: str*, *prepare_only: bool*)

Activity represents an action taken on a scheduling block.

An activity maps to a script that accomplishes the activity's goal. In a telescope control context, activities and goals could be 'allocate resources for this SB', 'observe this SB', etc. That is, users talk about doing something with the SB; their focus is not on which script needs to run and what script parameters are required to accomplish that task.

__init__(*activity_id: int*, *procedure_id: int | None*, *sbd_id: str*, *activity_name: str*, *prepare_only: bool*) → None

class ska_oso_oet.activity.domain.**ActivityState**(*value*)

ActivityState represent the state of an Activity.

ActivityState is currently a placeholder, to be elaborated with the full activity lifecycle (CREATED, RUNNING, SUCCEEDED, FAILED, etc.) in a later PI.

13.3 ska_oso_oet.activity.ui

The ska_oso_oet.activity.ui module contains code that belongs to the activity UI/presentation layer. This layer is the means by which external users or systems would interact with activities.

ska_oso_oet.activity.ui.**make_public_activity_summary**(*activity:* ActivitySummary)

Convert an ActivitySummary into JSON ready for client consumption.

The main use of this function is to replace the internal Activity ID with the resource URI, e.g., 1 -> http://localhost:5000/api/v1.0/procedures/1

Parameters
activity – ActivitySummary to convert

Returns
safe JSON representation

CHAPTER

FOURTEEN

SKA_OSO_OET.EVENT.TOPICS

class ska_oso_oet.event.topics.**activity**

Root topic for events related to activities.

class lifecycle

Topic for events related to activity lifecycle.

class running

Emitted when an activity starts running.

msgDataSpec(*request_id*, *result*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- result: ActivitySummary characterising the running activity

class pool

Topic for events on characterisation of the activity pool.

class list

Emitted when current activities and their status is enumerated.

msgDataSpec(*request_id*, *result*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- **result: list of ActivitySummary instances characterising** activites and their states.

class ska_oso_oet.event.topics.**procedure**

Root topic for events related to procedures.

class lifecycle

Topic for events related to procedure lifecycle.

class complete

Emitted when a Procedure has completed successfully and is no longer available to be called.

msgDataSpec(*request_id*, *result*)

- msg_src: ID of Procedure that completed

class created

Emitted when a procedure is created, i.e., a script is loaded and Python interpreter initialised.

msgDataSpec(*request_id*, *result*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request

- result: ProcedureSummary characterising the created procedure

class failed

Emitted when a procedure fails.

msgDataSpec(*request_id*, *result*)

- msg_src: component from which the event originated
- request_id: unique identifier for this event
- result: ProcedureSummary characterising the failed procedure

class stacktrace

Announces cause of a Procedure failure.

msgDataSpec(*stacktrace*)

- msg_src: component from which the request originated
- stacktrace: stacktrace as a string

class started

Emitted when any user function in a procedure is running, i.e., script init is called

msgDataSpec(*request_id*, *result*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- result: ProcedureSummary characterising the created procedure

class statechange

Emitted when a procedure status changes.

To be amalgamated and rationalised with other lifecycle events to better handle rerunnable scripts.

msgDataSpec(*new_state*)

- msg_src: component from which the request originated
- new_state: new state

class stopped

Emitted when a procedure stops, e.g., script completes or is aborted.

msgDataSpec(*request_id*, *result*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- result: ProcedureSummary characterising the created procedure

class pool

Topic for events on characterisation of the process pool.

class list

Emitted when current procedures and their status is enumerated.

msgDataSpec(*request_id*, *result*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- **result: list of ProcedureSummary instances characterising** procedures and their states.

class ska_oso_oet.event.topics.**request**

Root topic for events emitted when a user or system component has made a request.

class activity

Topic for user requests related to activities.

class list

Emitted when a request to enumerate all activities is received.

msgDataSpec(*request_id*, *activity_ids=None*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- activity_ids: Activity IDs to list.

class run

Emitted when a request to run an activity is received.

msgDataSpec(*request_id*, *cmd*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- cmd: ActivityCommand containing request parameters

class procedure

Topic for user requests related to procedures.

class create

Emitted when a request to create a procedure is received.

msgDataSpec(*request_id*, *cmd*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- cmd: PrepareProcessCommand containing request parameters

class list

Emitted when a request to enumerate all procedures is received.

msgDataSpec(*request_id*, *pids=None*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- pids: Procedure IDs to list

class start

Emitted when a request to start procedure execution is received.

msgDataSpec(*request_id*, *cmd*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- cmd: StartProcessCommand containing request parameters

class stop

Emitted when a request to stop a procedure is received.

msgDataSpec(*request_id*, *cmd*)

- msg_src: component from which the request originated
- request_id: unique identifier for this request
- cmd: StartProcessCommand containing request parameters

class ska_oso_oet.event.topics.**sb**

Root topic for events emitted relating to Scheduling Blocks

class lifecycle

Topic for events related to Scheduling Block lifecycle

class allocated

Emitted when resources have been allocated within SB execution

msgDataSpec(*sb_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID

class observation

Topic for events related to executing an observation within an SB

class finished

Emitted when an observation is finished

class failed

Emitted when an error was encountered during observation execution

msgDataSpec(*sb_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID

class succeeded

Emitted when an observation is finished successfully

msgDataSpec(*sb_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID

class started

Emitted when an observation is started

msgDataSpec(*sb_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID

class ska_oso_oet.event.topics.scan

Root topic for events emitted relating to Scans in the context of SB execution

class lifecycle

Topic for events related to SB scan lifecycle

class configure

Emitted when sub-array resources are configured for a scan

class complete

Emitted as scan configuration completes successfully.

msgDataSpec(*sb_id*, *scan_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID
- scan_id: Scan ID

class failed

Emitted if scan configuration fails.

msgDataSpec(*sb_id*, *scan_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID
- scan_id: Scan ID

class started

Emitted as scan configuration begins.

msgDataSpec(*sb_id*, *scan_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID
- scan_id: Scan ID

class end

Emitted when a scan finishes

class failed

Emitted when an error was encountered during a scan

msgDataSpec(*sb_id*, *scan_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID

class succeeded

Emitted when a scan completes successfully

msgDataSpec(*sb_id*, *scan_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID
- scan_id: Scan ID

class start

Emitted when resources have been allocated within SB execution

msgDataSpec(*sb_id*)

- msg_src: component from which the request originated
- sb_id: Scheduling Block ID
- scan_id: Scan ID

class ska_oso_oet.event.topics.**subarray**

Root topic for events emitted relating to individual Subarray activites

class configured

Emitted when subarray has been configured

msgDataSpec(*subarray_id*)

- msg_src: component from which the request originated
- sb_id: Subarray ID

class fault

Topic for events emitted when subarray cannot be reached

msgDataSpec(*subarray_id*, *error*)

- msg_src: component from which the request originated
- sb_id: Subarray ID

- error: Error response received from Subarray

class resources

Topic for events relating to Subarray resources

class allocated

Emitted when resources have been allocated to a subarray

msgDataSpec(*subarray_id*)

- msg_src: component from which the request originated
- sb_id: Subarray ID

class deallocated

Emitted when resources have been deallocated from a subarray

msgDataSpec(*subarray_id*)

- msg_src: component from which the request originated
- sb_id: Subarray ID

class scan

Topic for events emitted when a scan is run on subarray

class finished

Emitted when a scan is finished

msgDataSpec(*subarray_id*)

- msg_src: component from which the request originated
- sb_id: Subarray ID

class started

Emitted when a scan is started

msgDataSpec(*subarray_id*)

- msg_src: component from which the request originated
- sb_id: Subarray ID

class ska_oso_oet.event.topics.**user**

UNDOCUMENTED: created as parent without specification

class script

UNDOCUMENTED: created as parent without specification

class announce

UNDOCUMENTED: created without spec

msgDataSpec(*msg*)

- msg_src: component from which the request originated
- msg: user message

CHAPTER

FIFTEEN

SKA_OSO_OET.MPTOOLS

Top-level package for Multiprocessing Tools.

This package is substantially based on Pamela D McA'Nulty's mptools project, which is hosted at

https://github.com/PamelaM/mptools

Pamela presents an excellent article given an overview of the MPTools package at

https://www.cloudcity.io/blog/2019/02/27/things-i-wish-they-told-me-about-multiprocessing-in-python/

---

class ska_oso_oet.mptools.**EventMessage**(*msg_src: str*, *msg_type: str*, *msg: Any*)

EventMessage holds the message and message metadata for events sent on the event queue between MPTools ProcWorkers.

__init__(*msg_src: str*, *msg_type: str*, *msg: Any*)

class ska_oso_oet.mptools.**MPQueue**(*maxsize=0*, ***, *ctx*)

MPQueue is a multiprocessing Queue extended with convenience methods that return booleans to reflect success and failure rather than raising exceptions.

MPQueue adds methods to:

- get next item in an exception-free manner
- put an item in an exception-free manner

- drain queue to allow safe closure
- close queue in an exception-free manner

__init__(*maxsize=0*, *, *ctx*)

drain()

Drain all items from this MPQueue, yielding each item until all items have been removed.

safe_close() → int

Drain and close this MPQueue.

No more items can be added to this MPQueue one safe_close has been called.

safe_get(*timeout: float | None = 0.02*)

Remove and return an item from this MPQueue.

If optional arg timeout is None, safe_get returns an item if one is immediately available. If optional arg timeout is a positive number (the default), safe_get blocks at most timeout seconds for an item to become available. In either case, None is returned if no item is available.

Parameters
timeout – maximum timeout in seconds, or None for no waiting period

Returns
None if no item is available

safe_put(*item*, *timeout: float | None = 0.02*) → bool

Put an item on this MPQueue.

safe_put adds an item onto the queue if a free slot is available, blocking at most timeout seconds for a free slot and returning False if no free slot was available within that time.

Parameters

- **item** – item to add
- **timeout** – timeout in seconds

Returns
True if the operation succeeded within the timeout

class ska_oso_oet.mptools.**MainContext**(*mp_ctx: BaseContext | None = None*)

MainContext is the parent context for a set of worker processes that communicate via message queues.

MPQueue(**args*, ***kwargs*) → *MPQueue*

Create a new message queue managed by this context.

Parameters

- **args** – queue constructor args
- **kwargs** – queue constructor kwargs

Returns
message queue instance

Proc(*name: str*, *worker_class: Type[*ProcWorker*]*, **args*, ***kwargs*) → *Proc*

Create a new process managed by this context.

Parameters

- **name** – name for worker process
- **worker_class** – worker process class

- **args** – any worker class constructor args
- **kwargs** – any worker class constructor kwargs

Returns
worker instance

__init__(*mp_ctx: BaseContext | None = None*)

stop_procs() → Tuple[int, int]

Stop all ProcWorkers managed by this MPContext.

stop_procs requests cooperative shutdown of running ProcWorkers before escalating to more forceful methods using POSIX signals.

This function returns with a 2-tuple, the first item indicating the number of ProcWorkers that returned a non-zero exit status on termination, the second item indicating the number of ProcWorkers that required termination.

Returns
tuple of process termination stats

stop_queues() → int

Drain all queues, blocking until they have stopped.

Returns
number of items drained

class ska_oso_oet.mptools.**Proc**(*mp: BaseContext*, *name: str*, *worker_class: Type[*ProcWorker*]*, *shutdown_event: Event*, *event_q:* MPQueue, **args*, *logging_config: dict | None = None*, ***kwargs*)

Proc represents a child process of a MainContext.

Proc instances exist in the scope of a MainContext instance and in the same Python interpreter process as the MainContext. Procs are the MainContext's link to the ProcWorkers which run in separate Python interpreters. Every ProcWorker running in a child process is associated with one Proc.

Each Proc is responsible for bootstrapping its ProcWorker and managing its lifecycle. Proc arranges for an instance of the ProcWorker class passed as a constructor argument to be initialised and start running in a new child Python interpreter. Proc checks that the ProcWorker has started successfully by checking the status of a multiprocessing Event passed to the ProcWorker as a constructor argument, which should be set by the ProcWorker on successful startup. If ProcWorker startup does not complete successfully and the event is left unset, Proc will forcibly terminate the child process and report the error.

Proc is able to terminate its associated ProcWorker, first by giving the ProcWorker chance to co-operatively exit by setting the shutdown event. If the ProcWorker does not respond by exiting within the grace period set by Proc.SHUTDOWN_WAIT_SECS, Proc will forcibly terminate the ProcWorker's process.

Proc ensures that the shutdown event and MPQueues it receives are passed through to the ProcWorker. Note that by default only one shutdown event is created by the MainContext, so setting the shutdown event triggers shutdown in all ProcWorkers!

Proc does not contain any business logic or application-specific code, which should be contained in the ProcWorker - or more likely, a class that extends ProcWorker.

__init__(*mp: BaseContext*, *name: str*, *worker_class: Type[*ProcWorker*]*, *shutdown_event: Event*, *event_q:* MPQueue, **args*, *logging_config: dict | None = None*, ***kwargs*)

full_stop(*wait_time=3.0*) → None

Stop the ProcWorker child process.

The method will attempt to terminate ProcWorker execution, first by setting the shutdown event and giving the ProcWorker opportunity to cleanly exit. If the ProcWorker has not terminated after wait_time seconds, SIGTERM signals are sent to the child process hosting the ProcWorker.

Parameters
wait_time – grace time before sending SIGTERM signals

terminate(*max_retries=3*, *timeout=0.1*) → bool

Terminate the child process using POSIX signals.

This function sends SIGTERM to the child process, waiting timeout seconds before checking process status and, if the process is still alive, trying again.

Parameters

- **max_retries** – max retry attempts
- **timeout** – second to wait before retry

Returns
True if process termination was successful

class ska_oso_oet.mptools.**ProcWorker**(*name: str*, *startup_event: Event*, *shutdown_event: Event*, *event_q:* MPQueue, **args*, *logging_config: dict | None = None*, ***kwargs*)

ProcWorker is a template class for code that should execute in a child Python interpreter process.

ProcWorker contains the standard boilerplate code required to set up a well-behaved child process. It handles starting the process, connecting signal handlers, signalling the parent that startup completed, etc. ProcWorker does not contain any business logic, which should be defined in a subclass of ProcWorker.

The core ProcWorker template method is main_loop, which is called once startup is complete and main execution begins. In ProcWorker this method is left blank and should be overridden by the class extending ProcWorker. Once the main_loop method is complete, the ProcWorker is shut down.

MPTools provides some ProcWorker subclasses with main_loop implementations that provide different kinds of behaviour. For instance,

- TimerProcWorker.main_loop has code calls a function on a fixed cadence;
- QueueProcWorker.main_loop has code that gets items from a queue, calling a function with every item received.

__init__(*name: str*, *startup_event: Event*, *shutdown_event: Event*, *event_q:* MPQueue, **args*, *logging_config: dict | None = None*, ***kwargs*)

Create a new ProcWorker.

Parameters

- **name** – name of this worker
- **startup_event** – event to set on startup completion
- **shutdown_event** – event to monitor for shutdown
- **event_q** – queue for messages to/from MainWorker
- **args** –

init_signals() → *SignalObject*

Initialise signal handlers for this worker process.

Calling this method will install SIGTERM and SIGINT signal handlers for the running process.

static int_handler(*signal_object:* SignalObject, *exception_class*, *signal_num: int*, *current_stack_frame: frame | None*) → None

Custom signal handling function that requests co-operative ProcWorker shutdown by setting the shared Event, forcibly terminating the process by raising an instance of the given exception class if call limit has been exceeded.

Parameters

- **signal_object** – SignalObject to modify to reflect signal-handling state
- **exception_class** – Exception type to raise when call limit is exceeded
- **signal_num** – POSIX signal ID
- **current_stack_frame** – current stack frame

run() → int

Start ProcWorker execution.

This method performs the housekeeping required to set the worker instance running and starts the main loop. An exit code of 0 is returned if the main loop completes and exits cleanly.

Returns
exit status code

static term_handler(*signal_object:* SignalObject, *exception_class*, *signal_num: int*, *current_stack_frame: frame | None*) → None

Custom signal handling function that requests co-operative ProcWorker shutdown by setting the shared Event, forcibly terminating the process by raising an instance of the given exception class if call limit has been exceeded.

Parameters

- **signal_object** – SignalObject to modify to reflect signal-handling state
- **exception_class** – Exception type to raise when call limit is exceeded
- **signal_num** – POSIX signal ID
- **current_stack_frame** – current stack frame

class ska_oso_oet.mptools.**QueueProcWorker**(*name: str*, *startup_event: Event*, *shutdown_event: Event*, *event_q:* MPQueue, *work_q:* MPQueue, **args*, ***kwargs*)

QueueProcWorker is a ProcWorker that calls main_func with every item received on its work queue.

__init__(*name: str*, *startup_event: Event*, *shutdown_event: Event*, *event_q:* MPQueue, *work_q:* MPQueue, **args*, ***kwargs*)

Create a new QueueProcWorker.

The events and MPQueues passed to this constructor should be created and managed within the scope of a MainContext context manager and shared with other ProcWorkers, so that the communication queues are shared correctly between Python processes and there is a common event that can be set to notify all processes when shutdown is required.

Parameters

- **name** – name of this worker

- **startup_event** – event to trigger when startup is complete
- **shutdown_event** – event to monitor for shutdown
- **event_q** – outbox for posting messages to main context
- **work_q** – inbox message queue for work messages
- **args** – captures other anonymous arguments
- **kwargs** – captures other keyword arguments

main_loop() → None

main_loop delivers each event received on the work queue to the main_func template method, while checking for shutdown notifications.

Event delivery will cease when the shutdown event is set or a special sentinel message is sent.

class ska_oso_oet.mptools.**SignalObject**(*shutdown_event: Event*)

SignalObject is a struct holding properties and state referenced by mptools signal handlers during their processing.

Setting the SignalObject.shutdown_event will request all MPTools processes cooperatively shut down. SignalObject also records how many times a signal has been received, allowing escalation for processes that do not co-operate with shutdown_event requests.

__init__(*shutdown_event: Event*)

Create a new SignalObject.

Parameters

shutdown_event – shutdown Event shared between all MPTools processes

exception ska_oso_oet.mptools.**TerminateInterrupt**

class ska_oso_oet.mptools.**TimerProcWorker**(*name: str*, *startup_event: Event*, *shutdown_event: Event*, *event_q:* MPQueue, **args*, *logging_config: dict | None = None*, ***kwargs*)

TimerProcWorker is a ProcWorker that calls main_func on a fixed cadence.

ska_oso_oet.mptools.**default_signal_handler**(*signal_object:* SignalObject, *exception_class*, *signal_num: int*, *current_stack_frame: frame | None*) → None

Custom signal handling function that requests co-operative ProcWorker shutdown by setting the shared Event, forcibly terminating the process by raising an instance of the given exception class if call limit has been exceeded.

Parameters

- **signal_object** – SignalObject to modify to reflect signal-handling state
- **exception_class** – Exception type to raise when call limit is exceeded
- **signal_num** – POSIX signal ID
- **current_stack_frame** – current stack frame

ska_oso_oet.mptools.**init_signals**(*shutdown_event*, *int_handler*, *term_handler*) → *SignalObject*

Install SIGINT and SIGTERM signal handlers for the running Python process.

This function returns the SignalObject shared with signal handlers that the handlers use to store signal handling state.

Parameters

- **shutdown_event** – Event to set when SIGINT or SIGTERM is received

- **int_handler** – SIGINT handler function to install
- **term_handler** – SIGTERM handler function to install

Returns

SignalObject processed by signal handlers

ska_oso_oet.mptools.**proc_worker_wrapper**(*proc_worker_class: Type[ProcWorker]*, *name: str*, *startup_evt: Event*, *shutdown_evt: Event*, *event_q: MPQueue*, **args*, ***kwargs*)

This function is called to launch the worker task from within the child process.

Parameters

- **proc_worker_class** – worker class to instantiate
- **name** – name for this ProcWorker
- **startup_evt** – start-up event to share with worker
- **shutdown_evt** – shutdown event to share with worker
- **event_q** – event queue to share with worker
- **args** – any additional arguments to give to worker constructor

Returns

CHAPTER

SIXTEEN

SKA_OSO_OET.PROCEDURE

16.1 ska_oso_oet.procedure.application

The ska_oso_oet.procedure.application module holds classes and functionality that belong in the application layer of the OET. This layer holds the application interface, delegating to objects in the domain layer for business rules and actions.

class ska_oso_oet.procedure.application.**ArgCapture**(*fn: str*, *fn_args:* ProcedureInput, *time: float | None = None*)

ArgCapture is a struct to record function call and time of invocation.

__init__(*fn: str*, *fn_args:* ProcedureInput, *time: float | None = None*) → None

class ska_oso_oet.procedure.application.**PrepareProcessCommand**(*script:* ExecutableScript, *init_args:* ProcedureInput)

PrepareProcessCommand is input argument dataclass for the ScriptExecutionService prepare command. It holds all the information required to load and prepare a Python script ready for execution.

__init__(*script:* ExecutableScript, *init_args:* ProcedureInput) → None

class ska_oso_oet.procedure.application.**ProcedureHistory**(*process_states: List[Tuple[*ProcedureState*, float]] | None = None*, *stacktrace=None*)

ProcedureHistory is a non-functional dataclass holding execution history of a Procedure spanning all transactions.

process_states: records time for each change of ProcedureState (list of
tuples where tuple contains the ProcedureState and time when state was changed to)

stacktrace: None unless execution_error is True in which case stores
stacktrace from process

__init__(*process_states: List[Tuple[*ProcedureState*, float]] | None = None*, *stacktrace=None*)

class ska_oso_oet.procedure.application.**ProcedureSummary**(*id: int*, *script:* ExecutableScript, *script_args: List[*ArgCapture*]*, *history:* ProcedureHistory, *state:* ProcedureState)

ProcedureSummary is a brief representation of a runtime Procedure. It captures essential information required to describe a Procedure and to distinguish it from other Procedures.

__init__(*id: int*, *script:* ExecutableScript, *script_args: List[*ArgCapture*]*, *history:* ProcedureHistory, *state:* ProcedureState) → None

class ska_oso_oet.procedure.application.**ScriptExecutionService**(*mp_context: BaseContext | None = None*, *abort_script: ExecutableScript = FileSystemScript(script_uri='file:///home/docs/checkouts/readthedocs.org/user_builds/ska-telescope-ska-oso-oet/checkouts/5.2.0/src/ska_oso_oet/procedure/abort.py')*, *on_pubsub: List[Callable[[EventMessage], None]] | None = None*)

ScriptExecutionService provides the high-level interface and facade for the script execution domain (i.e., the 'procedure' domain).

The interface is used to load and run Python scripts in their own independent Python child process.

The shutdown method should be called to ensure cleanup of any multiprocessing artefacts owned by this service.

__init__(*mp_context: BaseContext | None = None*, *abort_script: ExecutableScript = FileSystemScript(script_uri='file:///home/docs/checkouts/readthedocs.org/user_builds/ska-telescope-ska-oso-oet/checkouts/5.2.0/src/ska_oso_oet/procedure/abort.py')*, *on_pubsub: List[Callable[[EventMessage], None]] | None = None*)

Create a new ScriptExecutionService.

The .stop() method of this ScriptExecutionService can run a second script once the current process has been terminated. By default, this second script calls SubArrayNode.abort() to halt further activities on the sub-array controlled by the terminated script. To run a different script, define the script URI in the abort_script_uri argument to this constructor.

Parameters

- **mp_context** – multiprocessing context to use or None for default
- **abort_script** – post-termination script for two-phase abort
- **on_pubsub** – callbacks to call when PUBSUB message is received

prepare(*cmd:* PrepareProcessCommand) → *ProcedureSummary*

Load and prepare a Python script for execution, but do not commence execution.

Parameters
cmd – dataclass argument capturing the script identity and load arguments

Returns

start(*cmd:* StartProcessCommand) → *ProcedureSummary*

Start execution of a prepared procedure.

Parameters
cmd – dataclass argument capturing the execution arguments

Returns

stop(*cmd:* StopProcessCommand) → List[*ProcedureSummary*]

Stop execution of a running procedure, optionally running a second script once the first process has terminated.

Parameters
cmd – dataclass argument capturing the execution arguments

Returns

summarise(*pids: List[int] | None = None*) → List[*ProcedureSummary*]

Return ProcedureSummary objects for Procedures with the requested IDs.

This method accepts an optional list of integers, representing the Procedure IDs to summarise. If the pids is left undefined, ProcedureSummary objects for all current Procedures will be returned.

Parameters
pids – optional list of Procedure IDs to summarise.

Returns
list of ProcedureSummary objects

class ska_oso_oet.procedure.application.**StartProcessCommand**(*process_uid: int*, *fn_name: str*, *run_args: ProcedureInput*, *force_start: bool = False*)

StartProcessCommand is the input argument dataclass for the ScriptExecutionService start command. It holds the references required to start a prepared script process along with any late-binding runtime arguments the script may require.

__init__(*process_uid: int*, *fn_name: str*, *run_args: ProcedureInput*, *force_start: bool = False*) → None

class ska_oso_oet.procedure.application.**StopProcessCommand**(*process_uid: int*, *run_abort: bool*)

StopProcessCommand is the input argument dataclass for the ScriptExecutionService Stop command. It holds the references required to Stop a script process along with any late-binding runtime arguments the script may require.

__init__(*process_uid: int*, *run_abort: bool*) → None

16.2 ska_oso_oet.procedure.domain

The ska_oso_oet.procedure.domain module holds domain entities from the script execution domain. Entities in this domain are things like scripts, OS processes, process supervisors, signal handlers, etc.

class ska_oso_oet.procedure.domain.**ExecutableScript**

Base class for all executable scripts.

Expected specialisations:

- scripts on filesystem
- scripts in git repository
- scripts given as a string
- scripts stored in the ODA
- etc.

__init__() → None

class ska_oso_oet.procedure.domain.**FileSystemScript**(*script_uri: str*)

Represents a script stored on the file system.

__init__(*script_uri: str*) → None

class ska_oso_oet.procedure.domain.**GitScript**(*script_uri: str*, *git_args: GitArgs*, *create_env: bool | None = False*)

Represents a script in a git repository.

__init__(*script_uri: str*, *git_args:* GitArgs, *create_env: bool | None = False*) → None

class ska_oso_oet.procedure.domain.**LifecycleMessage**(*msg_src: str*, *new_state:* ProcedureState)

LifecycleMessage is a message type for script lifecycle events.

__init__(*msg_src: str*, *new_state:* ProcedureState)

class ska_oso_oet.procedure.domain.**ModuleFactory**

Factory class used to return Python Module instances from a variety of storage back-ends.

static get_module(*script:* ExecutableScript)

Load Python code from storage, returning an executable Python module.

Parameters
script – Script object describing the script to load

Returns
Python module

class ska_oso_oet.procedure.domain.**ProcedureInput**(**args*, ***kwargs*)

ProcedureInput is a non-functional dataclass holding the arguments passed to a script method.

__init__(**args*, ***kwargs*)

class ska_oso_oet.procedure.domain.**ProcedureState**(*value*)

Represents the script execution state.

class ska_oso_oet.procedure.domain.**ProcessManager**(*mp_context: BaseContext | None = None*, *on_pubsub: List[Callable[[*EventMessage*], None]] | None = None*)

ProcessManager is the parent for all ScriptWorker processes.

ProcessManager is responsible for launching ScriptWorker processes and communicating API requests such as 'run main() function' or 'stop execution' to the running scripts. If a script execution process does not respond to the request, the process will be forcibly terminated. ProcessManager delegates to the mptools framework for process management functionality. Familiarity with mptools is useful in understanding ProcessManager functionality.

ProcessManager is also responsible for communicating script events to the rest of the system, such as events issued by the script or related to the script execution lifecycle.

It is recommended that ProcessManager.shutdown() be called before the ProcessManager is garbage collected. Failure to call shutdown could break the any multiprocessing state held in the scope of the manager or its child processes. This may or may not be a problem, depending on what is held and whether that state is used elsewhere. In short, be safe and call shutdown().

Note: ProcessManager does not maintain a history of script execution. History is recorded and managed by the ScriptExecutionService.

__init__(*mp_context: BaseContext | None = None*, *on_pubsub: List[Callable[[*EventMessage*], None]] | None = None*)

Create a new ProcessManager.

Functions passed in the on_pubsub argument will be called by the ProcessManager every time the ProcessManager's message loop receives a PUBSUB EventMessage. Callbacks should not perform significant processing on the same thread, as this would block the ProcessManager event loop.

Parameters

- **mp_context** – multiprocessing context use to create multiprocessing primitives

- **on_pubsub** – functions to call when a PUBSUB message is received

create(*script:* ExecutableScript, *, *init_args:* ProcedureInput) → int

Create a new Procedure that will, when executed, run the target Python script.

Objects that can only be shared through inheritance, such as multiprocessing object, can be shared by providing them as init_args here. These arguments will be provided to the init function in the user script, where present.

Parameters

- **script** – script URI, e.g. 'file://myscript.py'
- **init_args** – script initialisation arguments

Returns

run(*process_id: int*, *, *call: str*, *run_args:* ProcedureInput, *force_start: bool = False*) → None

Run a prepared Procedure.

This starts execution of the script prepared by a previous create() call.

Parameters

- **process_id** – ID of Procedure to execute
- **call** – name of function to call
- **run_args** – late-binding arguments to provide to the script
- **force_start** – Add run command to queue even if the script is not yet ready to run. Does not add command to queue if ProcedureState is FAILED, STOPPED, COMPLETE or UNKNOWN

Returns

stop(*process_id: int*) → None

Stop a running Procedure.

This stops execution of a currently running script.

Parameters
process_id – ID of Procedure to stop

Returns

class ska_oso_oet.procedure.domain.**ScriptWorker**(*name: str*, *startup_event: Event*, *shutdown_event: Event*, *event_q:* MPQueue, *work_q:* MPQueue, **args*, *scan_counter: Value | None = None*, *environment:* Environment | *None = None*, ***kwargs*)

ScriptWorker loads user code in a child process, running functions of that user code on request.

ScriptWorker acts when a message is received on its work queue. It responds to four types of messages:

1. LOAD - to load the specified code in this process
2. ENV - to install the dependencies for the specified script in this process
3. RUN - to run the named function in this process
4. PUBSUB - external pubsub messages that should be published locally

ScriptWorker converts external inter-process mptool pub/sub messages to intra-process pypubsub pub/sub messages. That is, EventMessages received on the local work queue are rebroadcast locally as pypubsub messages. Likewise, the ScriptWorker listens to all pypubsub messages broadcast locally, converts them to pub/sub EventQueue messages, and puts them on the 'main' queue for transmission to other interested ScriptWorkers.

__init__(*name: str*, *startup_event: Event*, *shutdown_event: Event*, *event_q:* MPQueue, *work_q:* MPQueue, **args*, *scan_counter: Value | None = None*, *environment:* Environment | *None = None*, ***kwargs*)

Create a new ProcWorker.

Parameters

- **name** – name of this worker
- **startup_event** – event to set on startup completion
- **shutdown_event** – event to monitor for shutdown
- **event_q** – queue for messages to/from MainWorker
- **args** –

main_loop() → None

main_loop delivers each event received on the work queue to the main_func template method, while checking for shutdown notifications.

Event delivery will cease when the shutdown event is set or a special sentinel message is sent.

publish_lifecycle(*new_state:* ProcedureState)

Broadcast a lifecycle status change event.

Parameters

new_state – new lifecycle state

republish(*topic: pubsub.pub.Topic = pubsub.pub.AUTO_TOPIC*, ***kwargs*) → None

Republish a local pypubsub event over the inter-process mptools event bus.

Parameters

- **topic** – message topic, set automatically by pypubsub
- **kwargs** – any metadata associated with pypubsub message

Returns

static term_handler(*signal_object*, *exception_class*, *signal_num: int*, *current_stack_frame*) → None

Custom signal handling function that simply raises an exception. Assuming the running Python script does not catch this exception, it will interrupt script execution and result in termination of that script.

We don't want all sibling script processes to terminate, hence no setting of shutdown_event is done in this handler.

Parameters

- **signal_object** – SignalObject to modify to reflect signal-handling state
- **exception_class** – Exception type to raise when call limit is exceeded
- **signal_num** – POSIX signal ID
- **current_stack_frame** – current stack frame

ska_oso_oet.procedure.domain.**script_signal_handler**(*signal_object*, *exception_class*, *signal_num: int*, *current_stack_frame*) → None

Custom signal handling function that simply raises an exception. Assuming the running Python script does not catch this exception, it will interrupt script execution and result in termination of that script.

We don't want all sibling script processes to terminate, hence no setting of shutdown_event is done in this handler.

Parameters

- **signal_object** – SignalObject to modify to reflect signal-handling state
- **exception_class** – Exception type to raise when call limit is exceeded
- **signal_num** – POSIX signal ID
- **current_stack_frame** – current stack frame

16.3 ska_oso_oet.procedure.environment

class ska_oso_oet.procedure.environment.**Environment**(*env_id: str*, *creating: <bound method BaseContext.Event of <multiprocessing.context.DefaultContext object at 0x7f485e1bb520>>*, *created: <bound method BaseContext.Event of <multiprocessing.context.DefaultContext object at 0x7f485e1bb520>>*, *location: str*, *site_packages: str*)

__init__(*env_id: str*, *creating: Event*, *created: Event*, *location: str*, *site_packages: str*) → None

16.4 ska_oso_oet.procedure.gitmanager

Static helper functions for cloning and working with a Git repository

class ska_oso_oet.procedure.gitmanager.**GitArgs**(*git_repo: str | None = 'https://gitlab.com/ska-telescope/oso/ska-oso-scripting.git'*, *git_branch: str | None = 'master'*, *git_commit: str | None = None*)

GitArgs captures information required to identify scripts located in git repositories.

__init__(*git_repo: str | None = 'https://gitlab.com/ska-telescope/oso/ska-oso-scripting.git'*, *git_branch: str | None = 'master'*, *git_commit: str | None = None*) → None

16.5 ska_oso_oet.procedure.ui

The ska_oso_oet.procedure.ui package contains code that belong to the OET procedure UI layer. This consists of the Procedure REST resources.

ska_oso_oet.procedure.ui.**create_procedure**()

Create a new Procedure.

This method requests creation of a new Procedure as specified in the JSON payload POSTed to this function.

Returns
JSON summary of created Procedure

ska_oso_oet.procedure.ui.**get_procedure**(*procedure_id: int*)

Get a Procedure.

This returns the Procedure JSON representation of the requested Procedure.

Parameters
procedure_id – ID of the Procedure to return

Returns
Procedure JSON

ska_oso_oet.procedure.ui.**get_procedures**()

List all Procedures.

This returns a list of Procedure JSON representations for all Procedures held by the service.

Returns
list of Procedure JSON representations

ska_oso_oet.procedure.ui.**make_public_procedure_summary**(*procedure:* ProcedureSummary)

Convert a ProcedureSummary into JSON ready for client consumption.

The main use of this function is to replace the internal Procedure ID with the resource URI, e.g., 1 -> http://localhost:5000/api/v1.0/procedures/1

Parameters
procedure – Procedure to convert

Returns
safe JSON representation

ska_oso_oet.procedure.ui.**update_procedure**(*procedure_id: int*)

Update a Procedure resource using the desired Procedure state described in the PUT JSON payload.

Parameters
procedure_id – ID of Procedure to modify

Returns
ProcedureSummary reflecting the final state of the Procedure

CHAPTER

SEVENTEEN

SKA_OSO_OET.UTILS

The ska_oso_oet.utils.ui module contains common helper code for the UI layers.

ska_oso_oet.utils.ui.**convert_request_dict_to_procedure_input**(*fn_dict: dict*) → *ProcedureInput*

Convert the dict of arguments for a single function into the domain.ProcedureInput

Parameters

fn_dict – Dict of the args and kwargs, eg {'args': [1, 2], 'kwargs': {'subarray_id': 42}}

Returns

The ProcedureInput, eg <ProcedureInput(1, 2, subarray_id=42)>

CHAPTER

EIGHTEEN

OBSERVATION EXECUTION TOOL

18.1 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.

18.2 Overview

The core of the OET is a script execution engine which runs a requested script in a child Python process. The engine supervises script execution, in that it can terminate the script at any time when requested, and captures the output and/or errors generated by the script for inspection by a (remote) client.

A REST layer makes the Python API for the script execution engine available via REST over HTTP. This project also contains a command line client to allow users to submit script execution requests to a remote OET backend.

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 *Module View: REST API*.
- The OET *OET command line tool* provides a Command Line Interface (CLI) to the OET backend.

More details on the OET architecture can be found in *C&C view: OET client and OET backend*.

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 ska-oso-scripting project.

18.3 Quickstart

Build a new OET image:

```
make oci-build
```

Execute the test suite and lint the project with:

```
make python-test
make python-lint
```

18.3.1 Format and lint on commit

We recommend you use pre-commit to automatically format and lint your commits. The commands below should be enough to get you up and running. Reference the official documentation for full installation details.

Pre-commit installation on Linux

```
# install pre-commit
sudo pip3 install pre-commit

# install git hook scripts
pre-commit install

# uninstall git hook scripts
pre-commit uninstall
```

Pre-commit installation on MacOS

The commands below were tested on MacOS 10.15.

```
# install pre-commit
pip3 install --user pre-commit

# install git hook scripts
~/Library/Python/3.8/bin/pre-commit install

# uninstall git hook scripts
~/Library/Python/3.8/bin/pre-commit uninstall
```

18.4 Makefile targets

This project extends the standard SKA Make targets with a few additional Make targets that can be useful for developers. These targets are:

Makefile target	Description
dev-up	deploy the OET using the current developer image, exposing REST ingress on the host
dev-down	tear down the developer OET deployment
rest	start the OET backend in a Docker container
diagrams	recreate PlantUML diagrams whose source has been modified
k8s-chart-test	run helm chart unit tests (note: requires helm unittest plugin: https://github.com/quintush/helm-unittest)
help	show a summary of the makefile targets above

18.5 Local development with k8s

OET REST server can be deployed locally using Helm and Kubernetes and OET CLI *OET command line tool* can be used to communicate with the server. OET CLI is installed as part of the Poetry virtual environment (see README) or can be used inside a running OET container/pod.

If using OET CLI within Poetry virtual environment these steps are needed for the CLI to access the REST server:

- set *rest.ingress.enabled* to *true* in *charts/ska-oso-oet/values.yaml*
- set *OET_REST_URI* environment variable with *export OET_REST_URI=http://<minikube IP>/<kube namespace>/ska-oso-oet/api/v1.0*

To deploy OET REST server run

```
make k8s-chart-install && make k8s-wait
```

18.6 Feature flags

OET feature flags are configured via environment variables and configuration files. The configuration file, ska_oso_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. ska_oso_oet.ini configuration file;
3. default flag value as specified in OET code.

No feature flags are available at this time.

PYTHON MODULE INDEX

S

INDEX

Symbols

A

C

D

E

F

G

I

L

M

N

P

T

U

V

W