
developer.skao.int Documentation

Release 0.5.4

Marco Bartolini

Mar 13, 2024

CONTENTS

1	Installation and Usage	3
2	Design and Best Practices	5
3	Configuration Schema	11
4	Configuration API	15
5	SDP command-line interface	37
6	Indices and tables	45
	Python Module Index	47
	Index	49

This repository contains the library for accessing SKA SDP configuration information. It provides ways for SDP controller and processing components to discover and manipulate the intended state of the system.

At the moment this is implemented on top of etcd, a highly-available database. This library provides primitives for atomic queries and updates to the stored configuration information.

INSTALLATION AND USAGE

1.1 Install with pip

```
pip install ska-sdp-config --extra-index-url https://artefact.skao.int/repository/pypi-  
↪internal/simple
```

1.2 Basic usage

Make sure you have a database backend accessible (etcd3 is supported at the moment). Location can be configured using the `SDP_CONFIG_HOST` and `SDP_CONFIG_PORT` environment variables. The defaults are `127.0.0.1` and `2379`, which should work with a local `etcd` started without any configuration.

You can find `etcd` pre-built binaries, for Linux, Windows, and macOS, here: <https://github.com/etcd-io/etcd/releases>.

You can also use `homebrew` to install `etcd` on macOS:

```
brew install etcd
```

If you encounter issues follow: <https://brewinstall.org/install-etcd-on-mac-with-brew/>

This should give you access to SDP configuration information, for instance try:

```
import ska_sdp_config  
  
config = ska_sdp_config.Config()  
  
for txn in config.txn():  
    for pb_id in txn.list_processing_blocks():  
        pb = txn.get_processing_block(pb_id)  
        print("{} ({}:{}".format(pb_id, pb.script['name'], pb.script['version']))
```

To read a list of currently active processing blocks with their associated scripts.

1.3 Command line

This package also comes with a command line utility for easy access to configuration data. For instance run:

SDP command-line interface

1.4 Running unit tests locally

You will need to have a database backend to run the tests as well. See “Basic usage” above for instructions on how to install an etcd backend on your machine.

Once you started the database (run etcd in the command line), you will be able to run the tests using pytest.

Alternative way is by using the two shell scripts in the scripts directory:

`docker_run_etcd.sh` -> Which runs etcd in a Docker container for testing the code. `docker_run_python.sh` -> Runs a python container and connects to the etcd instance.

Run the scripts from the root of the repository:

```
bash scripts/docker_run_etcd.sh
bash scripts/docker_run_python.sh
```

Once the container is started and mounted to the local directory.

Since the dependencies are managed by poetry, either run a poetry install, or pip install the repository (from the root):

```
pip install -e .
```

Then run the tests:

```
pytest tests/
```


DESIGN AND BEST PRACTICES

Quick points:

- Uses a key-value store
- Objects are represented as JSON
- Uses watchers on a key or range of keys to monitor for any updates

2.1 Transaction Basics

The SDP configuration database interface is built around the concept of transactions, i.e. blocks of read and write queries to the database state that are guaranteed to be executed atomically. For example, consider this code:

```
for txn in config.txn():
    a = txn.get('a')
    if a is None:
        txn.create('a', '1')
    else:
        txn.update('a', str(int(a)+1))
```

It is guaranteed that we increment the *'a'* key by exactly once here, no matter how many other processes might be operating on it. How does this work?

The way transactions are implemented follows the philosophy of [Software Transactional Memory](#) as opposed to a lock-based implementation. The idea is that all reads are performed, but all writes are actually delayed until the end of the transaction. So in the above example, *'a'* is actually read from the database using *'get'*, but the writes performed by *'create'* or *'update'* do not happen immediately.

Once the transaction finishes (the end of the *for* loop), *'commit'* sends a single request to the database that updates all written values **only if** none of the values that were read have been written in the meantime. If the commit fails, we repeat the transaction (that's why it is a loop!) until it succeeds. The idea is that this is fairly rare, and repeating the transaction should typically be cheap.

2.2 Usage Guidelines

What does this mean for everyday usage? Transactions should be as self-contained as possible - i.e. they should explicitly contain all assumptions about the database state they are making. If we wrote the above transaction as follows:

```
for txn in config.txn():
    a = txn.get('a')

for txn in config.txn():
    if a is None:
        txn.create('a', '1')
    else:
        txn.update('a', str(int(a)+1))
```

A whole number of things could happen between the first and the second transaction:

1. The 'a' key could not exist in the first transaction, but could have been created by the second (which would cause us to fail)
2. The 'a' key could exist in the first transaction, but could have been deleted by the second (which would also cause the above to fail)
3. Another transaction might have updated the 'a' key with a new value (which would cause that update to be lost)

A rule of thumb is that you should assume **nothing** about the database state at the start of a transaction. If you rely on something, you need to (re)query it after you enter it. If for some reason you couldn't merge the transactions above, you should write something like:

```
for txn in config.txn():
    a = txn.get('a')

for txn in config.txn():
    assert txn.get('a') == a, "database state independently updated!"
    if a is None:
        txn.create('a', '1')
    else:
        txn.update('a', str(int(a)+1))
```

This would especially catch case (3) above. This sort of approach can be useful when we want to make sub-transactions that only depend on a part of the overall state:

```
for txn in config.txn():
    keys = txn.list_keys('/as/')
    for key in keys:
        for txn in config.txn():
            a = txn.get(key)
            # Safety check: Path might have vanished in the meantime!
            if a is None:
                break
            # ... do something that depends solely on existence of "key" ...
```

This can especially be combined with watchers (see below) to keep track of many objects without requiring huge transactions.

2.3 Wrapping transactions

The safest way to work with transactions is to make them as “large” as possible, spanning all the way from getting inputs to writing outputs. This should be the default unless we have a strong reason to do it differently (examples for such reasons would be transactions becoming too large, or transactions taking so long that they never finish - but either should be extremely rare).

However, in the context of a program with complex behaviour this might appear cumbersome: This means we have to pass the transaction object to every single method that could either read or write the state. An elegant way to get around this is to move such methods to a “model” class that wraps the transaction itself:

```
def IncrementModel(Transaction):
    def __init__(self, txn):
        self._txn = txn
    def increase(key):
        a = self._txn.get(key)
        if a is None:
            self._txn.create(key, '1')
        else:
            self._txn.update(key, str(int(a)+1))

# ...
for txn in config.txn():
    model = IncrementModel(txn)
    model.increase('a')
```

In fact, we can provide factory functions that entirely hide the transaction object from view:

```
def increment_txn(config):
    for txn in config.txn():
        yield IncrementModel(txn)

# ...
for model in increment_txn(config):
    model.increase('a')
```

We could wrap this model the same way again to build as many abstraction layers as we want - key is that high-level methods such as “increase” are now directly tied to the existence of a transaction object.

2.4 Dealing with roll-backs

Especially as we start wrapping transactions more and more, we must keep in mind that while we can easily “roll back” any writes of the transaction (as they are not actually performed immediately), the same might not be true for program state. So for instance, the following would be unsafe:

```
to_update = ['a', 'b', 'c']
for model in increment_txn():
    while to_update:
        model.increase(to_update.pop())
```

Clearly this transaction would work differently the second time around! For this reason it is a good idea to keep in mind that while we expect the *for* to only execute once, it is entirely possible that they would execute multiple times, and the code should be written accordingly.

Fortunately, this sort of occurrence should be relatively rare - the following might be more typical:

```
objects_found = []
for model in increment_txn():
    for obj in model.list_objects():
        if model.some_check(obj):
            LOGGER.debug(f'Found {obj}!')
            objects_found.append(obj)
```

In this case, *objects_found* might contain duplicate objects if the transaction repeats - which could be easily fixed by moving the initialisation into the *for* loop.

On the other hand, note that transaction loops might also lead to duplicated log lines here, which might be seen as confusing. In this case, this is relatively benign and therefore likely acceptable. It might be possible to generate log messages at the start and end of transactions to make this more visible.

Another possible approach could be to replicate the transaction behaviour: for example, we could make the logging calls to *IncrementModel*, which would internally aggregate the logging lines to generate, which *increment_txn* could then emit in one go once the transaction actually goes through.

2.5 Watchers

Occasionally we might want to actively track something in the configuration. For sake of example, let's say we want to wait for a key to appear so we can print it. A simple implementation using polling might look like the following:

```
while True:
    for txn in config.txn():
        line = txn.get('/line_to_print')
        if line is not None:
            txn.delete('/line_to_print')
    if line is not None:
        print(line)
    time.sleep(1)
```

(Note that we are making sure to print outside the transaction loop - otherwise lines might get printed multiple times if we were running more than one instance of this program in parallel!)

But clearly this is not very good - it re-queries the database every second, which adds database load *and* is pretty slow. Instead, we can use a watcher loop:

```
for watcher in config.watcher():
    for txn in watcher.txn():
        line = txn.get('/line_to_print')
        if line is not None:
            txn.delete('/line_to_print')
    if line is not None:
        print(line)
```

Note that we are calling *txn* on the *watcher* instead of *config*: What is happening here is that the *watcher* object collects keys read by the transaction, and only iterates once one of them has been written. It is a concept that has a lot in common with the transaction loop, except that while the transaction loop only iterates if the transaction is inconsistent, the watcher loop *always* iterates.

Note that you can have multiple separate transactions within a watcher loop, which however are not guaranteed to be consistent. For example:

```

for watcher in config.watcher():
    for txn in watcher.txn():
        line = txn.get('/line_to_print')
    print('A:', line)
    for txn in watcher.txn():
        line = txn.get('/line_to_print')
    print('B:', line)

```

In this program we might get different results for *A* and *B*. However, the watcher *does* guarantee that the loop will iterate if any of the read values have been invalidated. So if the line was deleted between the two transaction, the following output would be generated:

```

A: something
B: None
A: None
B: None

```

After all, while transaction *B* had a current view of the situation the first time around, the view of transaction *A* became out-of-date.

By default, the watcher only iterates if any values read by a watcher transaction has changed. This may take an arbitrary amount of time (including infinite amount), hence we can “force” the watcher loop to go to its next iteration via two methods. A default *timeout* can be set either upon initiation:

```

for watcher in etcd3.watcher(timeout=60):
    ...

```

or manually with the `watcher.set_timeout(<new_timeout>)` method. The *timeout* is valid for the whole life-cycle of the watcher. Alternatively, you can set a “wake-up call”, on a loop-by-loop basis, using the `watcher.set_wake_up_at(<value_of_alarm>)` method. This guarantees that the watcher will wake up at the given time or earlier (specified as an absolute datetime object). This especially means that if the method gets called multiple times, the watcher will wake up at the earliest of the times specified, either by *timeout* or by any of the *wake_up* calls.

2.6 Etcd3 Backend Implementation

The backend uses the etcd3 client library, *python-etcd3* <<https://python-etcd3.readthedocs.io/en/latest/readme.html>>.

In this implementaion, all watchers share a common grpc connection to the etcd server for scalability. The values reported by the watchers are used to reconstruct a consistent database state. Progress notifications from the server are used to keep the watchers in sync and ensure that our view of the database state is consistent.

All watched keys in the database are cached and any keys which are in a steady state in each watcher iteration can skip the queue to the database. This is an efficient approach for any processes that watch a lot of keys.

2.7 Previous Etcd3 Backend Implementation

The backend was re-implemented in August 2023 to replace the client library *etcd3-py* with *python-etcd3*.

There were several issues with *etcd3-py* and workarounds and fixes had to be put in place to ensure the SDP Configuration Library kept working. As well as not being well maintained by its developers, *etcd3-py* uses a http connection per watcher and does not support caching.

As of August 2023, the old backend is still present in the repository and has been renamed to *etcd3_revolution1*. The new backend is set as the default. In order to use the old backend change the relevant environment variable before starting the etcd server.

```
# To use the previous backend
export SDP_CONFIG_BACKEND="etcd3revolution1"

# To use the current backend
export SDP_CONFIG_BACKEND="etcd3"
```

Or specify the desired backend:

```
import ska_sdp_config

config = ska_sdp_config.Config(backend="etcd3revolution1")
```

CONFIGURATION SCHEMA

This is the schema of the configuration database, effectively the control plane of the SDP.

3.1 Execution Block

Path /eb/[eb_id]

Dynamic state information of the execution block.

Contents:

```
{
  "eb_id": "eb-mvp01-20200425-000000",
  "max_length": 21600.0,
  "scan_types": [
    { "scan_type_id": "science", ... },
    { "scan_type_id": "calibration", ... }
  ],
  "pb_realtime": [ "pb-mvp01-20200425-000000", ... ],
  "pb_batch": [ ... ],
  "pb_receive_addresses": "pb-mvp01-20200425-000000",
  "current_scan_type": "science",
  "status": "SCANNING",
  "scan_id": 12345,
  "last_updated": "2022-08-01 10:01:12"
}
```

When the execution block is being executed, the status field is set to the observation state (obsState) of the subarray.
When the execution block is ended, status is set to FINISHED.

3.2 Processing Block

Path: /pb/[pb_id]

Static definition of processing block information.

Contents:

```
{
  "pb_id": "pb-mvp01-20200425-000000",
```

(continues on next page)

(continued from previous page)

```

    "eb_id": "eb-mvp01-20200425-000000",
    "script": {
        "kind": "realtime",
        "name": "vis_receive",
        "version": "0.2.0"
    },
    "parameters": { ... }
}

```

There are two kinds of processing, real-time and batch (offline). Real-time processing starts immediately, as it directly corresponds to an observation that is about to start. Batch processing will be inserted into a scheduling queue managed by the SDP, where it will typically be executed according to resource availability.

Valid kinds are `realtime` and `batch`. The script tag identifies the processing script version as well as the required underlying software (e.g. execution engines, processing components). `...` stands for arbitrary processing script-defined parameters.

3.2.1 Processing Block State

Path: `/pb/[pb_id]/state`

Dynamic state information of the processing block. If it does not exist, the processing block is still starting up.

Contents:

```

{
    "resources_available": True,
    "status": "RUNNING",
    "receive_addresses": [
        { "scan_type_id": "science", ... },
        { "scan_type_id": "calibration", ... },
    ],
    "last_updated": "2022-08-01 10:01:12"
}

```

Tracks the current state of the processing block. This covers both the SDP-internal state (as defined by the Execution Control Data Model) as well as information to publish via Tango for real-time processing, such as the status and receive addresses (for ingest).

3.2.2 Processing Block Owner

Path: `/pb/[pb_id]/owner`

Identifies the process executing the script. Used for leader election/lock as well as a debugging aid.

Contents:

```

{
    "command": [
        "vis_receive.py",
        "pb-mvp01-20200425-000000"
    ],
    "hostname": "pb-mvp01-20200425-000000-script-2kxfz",
}

```

(continues on next page)

(continued from previous page)

```
"pid": 1  
}
```


CONFIGURATION API

4.1 High-Level API

High-level API for SKA SDP configuration.

```
class ska_sdp_config.config.Config(backend=None, global_prefix: str = "", owner: dict = None,  
                                   component_name: str = None, wrapper: TxnWrapper | None = None,  
                                   **cargs)
```

Connection to SKA SDP configuration.

property alive_key: *str* | *None*

Get the alive key.

Returns

alive key or *None* if not set

property backend: *Backend*

Get the backend database object.

property client_lease: *Lease*

Return the lease associated with the client.

It will be kept alive until the client gets closed.

close() → *None*

Close the client connection.

is_alive() → *bool*

Is the connection alive in the sense that the keep-alive key exists?

Returns

whether it is

lease(ttl=10) → *Lease*

Generate a new lease.

Once entered can be associated with keys, which will be kept alive until the end of the lease. At that point a daemon thread will be started automatically to refresh the lease periodically (default seems to be TTL/4).

Parameters

ttl – Time to live for lease

Returns

lease object

set_alive() → *None*

Set the keep-alive key.

txn(*max_retries: int = 64*) → *Iterable[Transaction | TxnWrapper]*

Create a *Transaction* for atomic configuration query/change.

As we do not use locks, transactions might have to be repeated in order to guarantee atomicity. Suggested usage is as follows:

```
for txn in config.txn():
    # Use txn to read+write configuration
    # [Possibly call txn.loop()]
```

As the *for* loop suggests, the code might get run multiple times even if not forced by calling *Transaction.loop()*. Any writes using the transaction will be discarded if the transaction fails, but the application must make sure that the loop body has no other observable side effects.

See also *Usage Guide* for best practices for using transactions.

Parameters

max_retries – Number of transaction retries before a *RuntimeError* gets raised.

watcher(*timeout: float | None = None*) → *Iterable[Watcher]*

Create a new watcher.

Useful for waiting for changes in the configuration. Calling *Etcd3Watcher.txn()* on the returned watchers will create *Transaction* objects just like *txn()*.

See also *Usage Guide* for best practices for using watchers.

Parameters

timeout – Timeout for waiting. Watcher will loop after this time.

class *ska_sdp_config.config.Transaction*(*config: Any, txn: DbTransaction, paths: dict[str, str]*)

High-level configuration queries and updates to execute atomically.

create_controller(*state: dict*) → *None*

Create controller state.

Parameters

state – controller state

create_deployment(*dpl: Deployment*) → *None*

Request a change to cluster configuration.

Parameters

dpl – Deployment to add to database

create_deployment_state(*deploy_id: str, state: dict*) → *None*

Create Deployment state.

Parameters

- **deploy_id** – Deployment ID
- **state** – Deployment state to create

create_execution_block(*eb_id: str, state: dict*) → *None*

Create execution block.

Parameters

- **eb_id** – execution block ID
- **state** – execution block state

create_is_alive(*key*: *str*, *lease*: *Lease*) → *str*

Create an “is_alive” entry.

Parameters

- **key** – “is alive” key in database e.g. “lmc-controller/owner”
- **lease** – to associate with the entry

Returns

the full path of the entry

create_processing_block(*pblock*: *ProcessingBlock*) → *None*

Add a new ProcessingBlock to the configuration.

Parameters

pblock – Processing block to create

create_processing_block_state(*pb_id*: *str*, *state*: *dict*) → *None*

Create processing block state.

Parameters

- **pb_id** – Processing block ID
- **state** – Processing block state to create

create_script(*kind*: *str*, *name*: *str*, *version*: *str*, *script*: *dict*) → *None*

Create processing script definition.

Parameters

- **kind** – script kind
- **name** – script name
- **version** – script version
- **script** – script definition

create_subarray(*subarray_id*: *str*, *state*: *dict*) → *None*

Create subarray state.

Parameters

- **subarray_id** – subarray ID
- **state** – subarray state

delete_deployment(*dpl*: *Deployment*) → *None*

Undo a change to cluster configuration.

Parameters

dpl – Deployment to remove

delete_execution_block(*eb_id*: *str*, *recurse*: *bool* = *True*) → *None*

Delete an execution block (eb)

Parameters

- **eb_id** – Execution block ID

- **recurse** – if True, run recursive query and delete all objects

delete_processing_block(*pb_id*: *str*, *recurse*: *bool = True*) → *None*

Delete a processing block (pb)

Parameters

- **pb_id** – Processing block ID
- **recurse** – if True, run recursive query and delete all includes deleting /state and /owner of pb if exists

delete_script(*kind*: *str*, *name*: *str*, *version*: *str*) → *None*

Delete processing script definition.

Parameters

- **kind** – script kind
- **name** – script name
- **version** – script version

get_controller() → *dict* | *None*

Get controller state.

Returns

controller state

get_deployment(*deploy_id*: *str*) → *Deployment* | *None*

Retrieve details about a cluster configuration change.

Parameters

deploy_id – Name of the deployment

Returns

Deployment details

get_deployment_state(*deploy_id*: *str*) → *dict* | *None*

Get the current Deployment state.

Parameters

deploy_id – Deployment ID

Returns

Deployment state, or None if not present

get_execution_block(*eb_id*: *str*) → *dict*

Get execution block.

Parameters

eb_id – execution block ID

Returns

execution block state

get_processing_block(*pb_id*: *str*) → *ProcessingBlock* | *None*

Look up processing block data.

Parameters

pb_id – Processing block ID to look up

Returns

Processing block entity, or None if it doesn't exist

get_processing_block_owner(*pb_id: str*) → dict | None

Look up the current processing block owner.

Parameters

pb_id – Processing block ID to look up

Returns

Processing block owner data, or None if not claimed

get_processing_block_state(*pb_id: str*) → dict | None

Get the current processing block state.

Parameters

pb_id – Processing block ID

Returns

Processing block state, or None if not present

get_script(*kind: str, name: str, version: str*) → dict | None

Get processing script definition.

Parameters

- **kind** – script kind
- **name** – script name
- **version** – script version

Returns

script definition

get_subarray(*subarray_id: str*) → dict | None

Get subarray state.

Parameters

subarray_id – subarray ID

Returns

subarray state

is_alive(*key: str*) → bool

Check if the “is alive” key still exists.

Parameters

key – “is alive” key in database e.g. “lmc-controller/owner”

Returns

True if it does

is_processing_block_owner(*pb_id: str*) → bool

Check whether this client is owner of the processing block.

Parameters

pb_id – Processing block ID to look up

Returns

Whether processing block exists and is claimed

list_deployments(*prefix: str = ""*) → list[str]

List all current deployments.

Returns

Deployment IDs

list_execution_blocks(*prefix: str = ""*) → list[str]

Query execution block IDs from the configuration.

Parameters

prefix – if given, only search for execution block IDs with the given prefix

Returns

execution block IDs, in lexicographical order

list_processing_blocks(*prefix: str = ""*) → list[str]

Query processing block IDs from the configuration.

Parameters

prefix – If given, only search for processing block IDs with the given prefix

Returns

Processing block ids, in lexicographical order

list_scripts(*kind: str = "", name: str = ""*) → list[tuple[str]]

List processing script definitions.

Parameters

- **kind** – script kind. Default empty
- **name** – script name. Default empty

Returns

list of script definitions

list_subarrays(*prefix: str = ""*) → list[str]

Query subarray IDs from the configuration.

Parameters

prefix – if given, only search for subarray IDs with the given prefix

Returns

subarray IDs, in lexicographical order

loop(*wait: bool = False, timeout: float | None = None*) → None

Repeat transaction regardless of whether commit succeeds.

Parameters

- **wait** – If transaction succeeded, wait for any read values to change before repeating it.
- **timeout** – Maximum time to wait, in seconds

new_execution_block_id(*generator: str*) → str

Generate a new execution block ID that is not yet in use.

Parameters

generator – Name of the generator

Returns

execution block ID

new_processing_block_id(*generator: str*) → str

Generate a new processing block ID that is not yet in use.

Parameters

generator – Name of the generator

Returns

Processing block ID

property raw: **DbTransaction**

Return transaction object for accessing database directly.

take_processing_block(*pb_id: str, lease: Lease*) → *None*

Take ownership of the processing block.

Parameters

- **pb_id** – Processing block ID to take ownership of
- **lease** – lease

Raises

backend.ConfigCollision

update_controller(*state: dict*) → *None*

Update controller state.

Parameters

state – controller state

update_deployment_state(*deploy_id: str, state: dict*) → *None*

Update Deployment state.

Parameters

- **deploy_id** – Deployment ID
- **state** – Deployment state to update

update_execution_block(*eb_id: str, state: dict*) → *None*

Update execution block.

Parameters

- **eb_id** – execution block ID
- **state** – execution block state

update_processing_block(*pblock: ProcessingBlock*) → *None*

Update a ProcessingBlock in the configuration.

Parameters

pblock – Processing block to update

update_processing_block_state(*pb_id: str, state: dict*) → *None*

Update processing block state.

Parameters

- **pb_id** – Processing block ID
- **state** – Processing block state to update

update_script(*kind: str, name: str, version: str, script: dict*) → *None*

Update processing script definition.

Parameters

- **kind** – script kind
- **name** – script name
- **version** – script version
- **script** – script definition

update_subarray(*subarray_id*: *str*, *state*: *dict*) → *None*

Update subarray state.

Parameters

- **subarray_id** – subarray ID
- **state** – subarray state

ska_sdp_config.config.dict_to_json(*obj*: *dict*) → *str*

Format a dictionary for writing it into the database.

Parameters

obj – Dictionary object to format

Returns

String representation

4.2 Entities

4.2.1 Processing Block

Processing block configuration entities.

class ska_sdp_config.entity.pb.**ProcessingBlock**(*pb_id*, *eb_id*, *script*, *parameters*=*None*,
dependencies=*None*, ***kwargs*)

Processing block entity.

Collects configuration information relating to a processing job for the SDP. This might be either real-time (supporting a running observation) or batch (to process data after the fact).

Actual execution of processing steps will be performed by a (parameterised) processing script interpreting processing block information.

property dependencies

Return dependencies on other processing blocks.

property eb_id

Return execution block instance ID, if associated with one.

property parameters

Return processing script-specific parameters.

property pb_id

Return the processing block ID.

property script

Return information identifying the processing script.

to_dict()

Return data as dictionary.

4.2.2 Deployment

Deployment configuration entities.

class `ska_sdp_config.entity.deployment.Deployment(dpl_id, kind, args)`

Deployment entity.

Collects configuration information relating to a cluster configuration change.

property `args`

Return deployment arguments.

property `dpl_id`

Return the deployment id.

property `kind`

Return deployment kind.

to_dict()

Return data as dictionary.

4.3 Backends

4.3.1 Common

Common functionality for implementing backends.

exception `ska_sdp_config.backend.common.ConfigCollision(path: str, message: str)`

Exception generated if key to create already exists.

exception `ska_sdp_config.backend.common.ConfigVanished(path: str, message: str)`

Exception generated if key to update that does not exist.

`ska_sdp_config.backend.common.depth_of_path(path: str) → int`

Get the depth of a path, this is the number of “/” in it.

Returns

the depth

4.3.2 Etcd3 backend

Etcd3 backend for SKA SDP configuration DB.

class `ska_sdp_config.backend.etcd3.Etcd3Backend(host='localhost', port='2379', max_retries: int = 15, retry_time: float = 0.1, **kw_args)`

Highly consistent database backend store.

See <https://github.com/kagniz/python-etcd3>

close() → None

Close the client connection.

create(*path*: *str*, *value*: *str*, *lease*: *etcd3.Lease* | *None* = *None*) → *None*

Create a key and initialise it with the value.

Parameters

- **path** – Path to create
- **value** – Value to set
- **lease** – Lease to associate

Raises

ConfigCollision if the key already exists

delete(*path*: *str*, *must_exist*: *bool* = *True*, *recursive*: *bool* = *False*, *prefix*: *bool* = *False*, *max_depth*: *int* = 16)

Delete the given key or key range.

Parameters

- **path** – path (prefix) of keys to remove
- **must_exist** – Fail if path does not exist?
- **recursive** – Delete children keys at lower levels recursively
- **max_depth** – Recursion limit
- **prefix** – Delete all keys at given level with prefix

get(*path*: *str*, *revision*: *DbRevision* | *None* = *None*) → *tuple*[*str*, *DbRevision*]

Get value of a key.

Parameters

- **path** – Path of key to query
- **revision** – to get

Returns

value and revision

lease(*ttl*: *float* = 10) → *Lease*

Generate a new lease.

Once entered, it can be associated with keys which will be kept alive until the end of the lease.

Note that this involves starting a daemon thread that will refresh the lease periodically (default seems to be TTL/4).

Parameters

ttl – Time to live for lease

Returns

lease object

list_keys(*path*: *str*, *recurse*: *int* | *Iterable*[*int*] = 0, *revision*: *DbRevision* | *None* = *None*, *with_values*: *bool* = *False*) → *tuple*[*list*[*str*], *DbRevision*]

List keys under given path.

Parameters

- **path** – Prefix of keys to query. Append ‘/’ to list child paths.
- **recurse** – Maximum recursion level to query. If iterable, cover exactly the recursion levels specified.

- **revision** – Database revision for which to list
- **with_values** – Also return key values and mod revisions (i.e. sorted list of key-value-rev tuples)

Returns

(sorted key list, DbRevision object)

txn(*max_retries*: *int* = 64) → *Iterable*[*Etcd3Transaction*]

Create a new transaction.

Note that this uses an optimistic STM-style implementation, which cannot guarantee that a transaction runs through successfully. Therefore, this function returns an iterator, which loops until the transaction succeeds:

```
for txn in etcd3.txn():
    # ... transaction steps ...
```

Note that this will in most cases only execute one iteration. If you actually want to loop - for instance because you intend to wait for something to happen in the configuration - use *watcher()* instead.

Parameters

max_retries – Maximum number of transaction loops

Returns

Transaction iterator

update(*path*: *str*, *value*: *str*) → *None*

Update an existing key. Fails if the key does not exist.

Parameters

- **path** – Path to update
- **value** – New value of key

Raises

ConfigVanished if the key does not exist

watcher(*timeout*=*None*, *txn_wrapper*: *Callable*[[*'Etcd3Transaction'*], *object*] = *None*, *reqquery_progress*: *float* = 0.2) → *Iterable*[*Etcd3Watcher*]

Create a new watcher.

Useful for waiting for changes in the configuration. See *Etcd3Watcher*.

Parameters

- **timeout** – Timeout for waiting. Watcher will loop after this time.
- **txn_wrapper** – Function to wrap transactions returned by the wrapper.
- **reqquery_progress** – How often we “refresh” the current database state for watcher transactions even without watcher notification (upper bound on how “stale” non-watched values retrieved in transactions can be)

Returns

Watcher iterator

class *ska_sdp_config.backend.etcd3.Etcd3Transaction*(*backend*: *Etcd3Backend*, *client*: *etcd3.client*, *max_retries*: *int* = 64)

A series of queries and updates to be executed atomically.

commit() → *bool*

Commit the transaction to the database.

This can fail, in which case the transaction must get *reset* and built again.

Returns

Whether the commit succeeded

create(*path*: *str*, *value*: *str*, *lease*: *etcd3.Lease* | *None* = *None*) → *None*

Create a key and initialise it with the value.

Fails if the key already exists. If a lease is given, the key will automatically get deleted once it expires.

Parameters

- **path** – Path to create
- **value** – Value to set
- **lease** – Lease to associate

Raises

ConfigCollision if the key already exists

delete(*path*: *str*, *must_exist*: *bool* = *True*, *recursive*: *bool* = *False*, *max_depth*: *int* = 16, *prefix*: *bool* = *False*) → *None*

Delete the given key.

Parameters

- **path** – Path of key to remove
- **must_exist** – Fail if path does not exist?
- **recursive** – Delete children keys at lower levels recursively (not used yet)

get(*path*: *str*) → *str* | *None*

Get value of a key.

Parameters

path – Path of key to query

Returns

Key value. None if it doesn't exist.

list_keys(*path*: *str*, *recurse*: *int* | *Iterable[int]* = 0) → *list[str]*

List keys under given path.

Parameters

- **path** – Prefix of keys to query. Append '/' to list child paths.
- **recurse** – Children depths to include in search

Returns

sorted key list

on_commit(*callback*: *Callable[[], None]*) → *None*

Register a callback to call when the transaction succeeds.

Exists mostly to enable test cases.

Parameters

callback – Callback to call

reset(*revision*: *DbRevision* | *None* = *None*) → *None*

Reset the transaction, so it can be restarted after commit().

Parameters

revision – to reset

Raises

RuntimeError if the transaction is not committed.

property revision: *int*

The last-committed database revision.

Only valid to call after the transaction has been committed.

Returns

revision from *DbRevision*

update(*path*: *str*, *value*: *str*) → *None*

Update an existing key.

Fails if the key does not exist.

Parameters

- **path** – Path to update
- **value** – Value to set

Raises

ConfigVanished if the key is not found

4.3.3 Etcd3 backend revolution 1

Etcd3 backend for SKA SDP configuration DB, using client from <https://github.com/Revolution1/etcd3-py>

```
class ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1(*args, max_retries: int
                                                                    = 15, retry_time: float
                                                                    = 0.1, **kw_args)
```

Highly consistent database backend store.

See <https://github.com/etcd-io/etcd>

All parameters will be passed on to `etcd3.Client()`.

close() → *None*

Close the client connection.

create(*path*: *str*, *value*: *str*, *lease*: *etcd3_revolution1.Lease* = *None*) → *None*

Create a key and initialise it with the value.

Fails if the key already exists. If a lease is given, the key will automatically get deleted once it expires.

Parameters

- **path** – Path to create
- **value** – Value to set
- **lease** – Lease to associate

Raises

ConfigCollision

delete(*path*: *str*, *must_exist*: *bool* = *True*, *recursive*: *bool* = *False*, *prefix*: *bool* = *False*, *max_depth*: *int* = 16) → *None*

Delete the given key or key range.

Parameters

- **path** – Path (prefix) of keys to remove
- **must_exist** – Fail if path does not exist?
- **recursive** – Delete children keys at lower levels recursively
- **prefix** – Delete all keys at given level with prefix
- **max_depth** – Recursion limit

Returns

Whether transaction was successful

get(*path*: *str*, *revision*: *DbRevision* | *None* = *None*) → *tuple*[*str*, *DbRevision*]

Get value of a key.

Parameters

- **path** – Path of key to query
- **revision** – Database revision for which to read key

Returns

(value, revision). value is *None* if it doesn't exist

lease(*ttl*: *int* = 10) → *Lease*

Generate a new lease.

Once entered can be associated with keys, which will be kept alive until the end of the lease. Note that this involves starting a daemon thread that will refresh the lease periodically (default seems to be TTL/4).

Parameters

ttl – Time to live for lease

Returns

lease object

list_keys(*path*: *str*, *recurse*: *int* = 0, *revision*: *DbRevision* | *None* = *None*) → *tuple*[*list*[*str*], *DbRevision*]

List keys under given path.

Parameters

- **path** – Prefix of keys to query. Append '/' to list child paths.
- **recurse** – Maximum recursion level to query. If iterable, cover exactly the recursion levels specified.
- **revision** – Database revision for which to list

Returns

(sorted key list, revision)

txn(*max_retries*: *int* = 64) → *Iterable*[*Etcd3Transaction*]

Create a new transaction.

Note that this uses an optimistic STM-style implementation, which cannot guarantee that a transaction runs through successfully. Therefore, this function returns an iterator, which loops until the transaction succeeds:


```
for txn in etcd3.txn():
    # ... transaction steps ...
```

Note that this will in most cases only execute one iteration. If you actually want to loop - for instance because you intend to wait for something to happen in the configuration - use `watcher()` instead.

Parameters

max_retries – Maximum number of transaction loops

Returns

Transaction iterator

update(*path*: *str*, *value*: *str*, *must_be_rev*: *DbRevision* | *None* = *None*) → *None*

Update an existing key. Fails if the key does not exist.

Parameters

- **path** – Path to update
- **value** – Value to set
- **must_be_rev** – Fail if found value does not match given revision (atomic update)

Raises

`ConfigVanished`

watch(*path*: *str*, *prefix*: *bool* = *False*, *revision*: *DbRevision* | *None* = *None*, *depth*: *int* | *None* = *None*)

Watch key or key range.

Use a path ending with `'/'` in combination with *prefix* to watch all child keys.

Parameters

- **path** – Path of key to query, or prefix of keys.
- **prefix** – Watch for keys with given prefix if set
- **revision** – Database revision from which to watch
- **depth** – tag depth

Returns

`Etcd3Watch` object for watch request

watcher(*timeout*: *float* = *None*, *txn_wrapper*: *TxnWrapper* = *None*) → *Iterable*[`Etcd3Watcher`]

Create a new watcher.

Useful for waiting for changes in the configuration. See `Etcd3Watcher`.

Parameters

- **timeout** – Timeout for waiting. Watcher will loop after this time.
- **txn_wrapper** – Function to wrap transactions returned by the wrapper.

Returns

Watcher iterator

```
class ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction(backend:
    Etcd3BackendRevolution1,
    client: etcd3_revolution1.Client,
    max_retries: int = 64)
```

A series of queries and updates to be executed atomically.

Use `Etcd3Backend.txn()` or `Etcd3Watcher.txn()` to construct transactions.

clear_watch() → `None`

Stop all currently active watchers.

Deprecated: Use `Etcd3Watcher` instead.

commit() → `bool`

Commit the transaction to the database.

This can fail, in which case the transaction must get *reset* and built again.

Returns

Whether the commit succeeded

create(*path*: `str`, *value*: `str`, *lease*: `etcd3_revolution1.Lease = None`) → `None`

Create a key and initialise it with the value.

Fails if the key already exists. If a lease is given, the key will automatically get deleted once it expires.

Parameters

- **path** – Path to create
- **value** – Value to set
- **lease** – Lease to associate

Raises

`ConfigCollision`

delete(*path*: `str`, *must_exist*: `bool = True`, *recursive*: `bool = False`) → `None`

Delete the given key.

Parameters

- **path** – Path of key to remove
- **must_exist** – Fail if path does not exist?
- **recursive** – Delete children keys at lower levels recursively (not used)

get(*path*: `str`) → `str`

Get value of a key.

Parameters

path – Path of key to query

Returns

Key value. `None` if it doesn't exist.

list_keys(*path*: `str`, *recurse*: `int = 0`) → `list[str]`

List keys under given path.

Parameters

- **path** – Prefix of keys to query. Append `'/'` to list child paths.
- **recurse** – Children depths to include in search

Returns

sorted key list

loop(*watch*: *bool* = *False*, *watch_timeout*: *float* | *None* = *None*)

Repeat transaction execution, even if it succeeds.

Deprecated: Use [Etcd3Watcher](#) instead, or loop manually.

Parameters

- **watch** – Once the transaction succeeds, block until one of the values read changes, then loop the transaction
- **watch_timeout** – timeout value

on_commit(*callback*: *Callable*[[], *None*] → *None*)

Register a callback to call when the transaction succeeds.

A bit of a hack, but occasionally useful to add additional side-effects to a transaction that are guaranteed to not get duplicated.

Parameters

callback – Callback to call

reset(*revision*: *DbRevision* | *None* = *None*) → *None*

Reset the transaction, so it can be restarted after commit().

property revision: *int*

The last-committed database revision.

Only valid to call after the transaction has been committed.

trigger_loop() → *None*

Manually triggers a loop

Effectively makes loop(True) behave like loop(False), looping immediately. This is useful for interrupting a blocking watch() from a different thread.

update(*path*: *str*, *value*: *str*) → *None*

Update an existing key. Fails if the key does not exist.

Parameters

- **path** – Path to update
- **value** – Value to set

Raises

ConfigVanished

watch() → *None*

Wait for a change on one of the values read.

Deprecated: Use [Etcd3Watcher](#) instead.

Returns

The revision at which a change was detected.

class `ska_sdp_config.backend.etcd3_revolution1.Etcd3Watch`(*backend*: [Etcd3BackendRevolution1](#),
tagged_path: *str*, *start_revision*: *int*,
prefix: *bool*, *max_retries*: *int* = 20,
retry_time: *float* = 0.1)

Wrapper for etc3 watch requests.

Entering the watcher using a *with* block yields a queue of (*key*, *val*, *rev*) triples.

start(*queue: Queue = None*) → *None*

Activates the watcher, yielding a queue for updates.

stop()

Deactivates the watcher.

class `ska_sdp_config.backend.etcd3_revolution1.Etcd3Watcher`(*backend: Etcd3BackendRevolution1*,
client: etcd3_revolution1.Client,
timeout: float = None, *txn_wrapper: TxnWrapper = None*)

Watch for database changes by using nested transactions

Use as follows:

```
for watcher in config.watcher():
    for txn in watcher.txn():
        # ... do something
    for txn in watcher.txn():
        # ... do something else
```

At the end of a for loop iteration, the watcher will start watching all values read by transactions started through `txn()`, and only repeat the execution of the loop body once one of these values has changed.

trigger() → *None*

Manually triggers a loop

Can be called from a different thread to force a loop, even if the watcher is currently waiting.

txn(*max_retries: int = 64*) → *Iterable[Etcd3Transaction | TxnWrapper]*

Create nested transaction.

The watcher loop will iterate when any value read by transactions created by this method have changed in the database.

Note that these transactions otherwise behave exactly as normal transactions: As long as they are internally consistent, they will be committed. This means there is no consistency guarantees between transactions created from the same watcher, i.e. one transaction might read one value from the database while a later one reads another.

Parameters

max_retries – Maximum number of times the transaction will be tried before giving up.

4.3.4 Etcd3 watcher

4.3.5 Memory backend

Memory backend for SKA SDP configuration DB.

The main purpose of this is for use in testing. In principle, it should behave in the same way as the etcd backend. No attempt has been made to make it thread-safe, so it probably isn't.

class `ska_sdp_config.backend.memory.MemoryBackend`

In-memory backend implementation, principally for testing.

close() → *None*

Close the resource. This does nothing.

create(*path*: *str*, *value*: *str*, *lease*: *Lease* | *None* = *None*) → *None*

Create a key and initialise it with the value.

Parameters

- **path** – Path to create
- **value** – Value to set
- **lease** – Lease to associate

Raises

ConfigCollision if the key already exists

delete(*path*: *str*, *must_exist*: *bool* = *True*, *recursive*: *bool* = *False*, *prefix*: *bool* = *False*, *max_depth*: *int* = *16*) → *None*

Delete the given key or key range.

Parameters

- **path** – path (prefix) of keys to remove
- **must_exist** – Fail if path does not exist?
- **recursive** – Delete children keys at lower levels recursively
- **max_depth** – Recursion limit
- **prefix** – Delete all keys at given level with prefix

get(*path*: *str*, *revision*: *DbRevision* | *None* = *None*) → *tuple*[*str*, *DbRevision*]

Get value of a key.

Parameters

- **path** – Path of key to query
- **revision** – to get

Returns

value and revision

lease(*ttl*: *float* = *10*) → *Lease*

Generate a dummy lease object.

Parameters

- **ttl** – time to live

Returns

dummy lease object

list_keys(*path*: *str*, *recurse*: *int* = *0*) → *list*[*str*]

Get a list of the keys at the given path.

In common with the etcd backend, the structure is “flat” rather than a real hierarchy, even though it looks like one.

Parameters

- **path** – prefix of keys to query
- **recurse** – maximum recursion level to query

Returns

list of keys

txn(*max_retries*: *int* = 64) → *Iterable*[*MemoryTransaction*]

Create an in-memory “transaction”.

Parameters

max_retries – Maximum number of transaction loops

Returns

transaction object

update(*path*: *str*, *value*: *str*) → *None*

Update an existing key. Fails if the key does not exist.

Parameters

- **path** – Path to update
- **value** – New value of key

Raises

ConfigVanished if the key does not exist

watcher(*timeout*: *float* = *None*, *txn_wrapper*: *TxnWrapper* = *None*) → *Watcher*

Create an in-memory “watcher”.

Parameters

- **timeout** – timeout in seconds
- **txn_wrapper** – wrapper (factory) to return transaction

Returns

MemoryWatcher object (mock of *Etcd3Watcher*)

class *skao_sdp_config.backend.memory.MemoryTransaction*(*backend*: *Backend*)

Transaction wrapper around the backend implementation.

Transactions always succeed if they are valid, so there is no need to loop; however the iterator is supported for compatibility with the etcd backend.

commit() → *bool*

Commit the transaction. This does nothing.

create(*path*: *str*, *value*: *str*, *lease*: *Lease* | *None* = *None*) → *None*

Create a key and initialise it with the value.

Fails if the key already exists. If a lease is given, the key will automatically get deleted once it expires.

Parameters

- **path** – Path to create
- **value** – Value to set
- **lease** – Lease to associate

Raises

ConfigCollision if the key already exists

delete(*path*: *str*, *must_exist*: *bool* = *True*, *recursive*: *bool* = *False*)

Delete the given key.

Parameters

- **path** – Path of key to remove

- **must_exist** – Fail if path does not exist?
- **recursive** – Delete children keys at lower levels recursively (not used yet)

get(*path: str*) → *str*

Get value of a key.

Parameters

path – Path of key to query

Returns

Key value or None if it doesn't exist.

list_keys(*path: str, recurse: int = 0*) → *list[str]*

List keys under given path.

Parameters

- **path** – Prefix of keys to query. Append '/' to list child paths.
- **recurse** – Children depths to include in search

Returns

sorted key list

loop(**_args*, ***_kwargs*) → *None*

Loop the transaction. This does nothing.

reset(*revision: DbRevision | None = None*) → *None*

Reset the transaction. This does nothing.

update(*path: str, value: str*) → *None*

Update an existing key.

Fails if the key does not exist.

Parameters

- **path** – Path to update
- **value** – Value to set

Raises

ConfigVanished if the key is not found

class `ska_sdp_config.backend.memory.MemoryWatcher`(*backend: Backend, timeout: float | None = None, txn_wrapper: TxnWrapper = None*)

Watcher wrapper around the backend implementation (Etcd3Watcher).

txn() → *Iterable[MemoryTransaction]*

Yield the wrapped MemoryTransaction object.

It does not implement the commit check that is part of Etcd3Watcher.txn(), hence it acts as MemoryBackend.txn()

SDP COMMAND-LINE INTERFACE

Command Line Interface: `ska-sdp`

To run the CLI, you must start a shell in the console pod (assuming you have SDP deployed in Kubernetes/Minikube, for instructions follow: [SDP standalone](#)).

```
kubectl exec -it ska-sdp-console-0 -n <namespace> -- bash
```

Once in, to access the help window of `ska-sdp`, run:

```
ska-sdp -h
```

5.1 Command - SDP Object matrix

This is a table/matrix of the existing commands of `ska-sdp` and what they can do with a specific SDP Object.

Commands:

- list
- get/watch
- create
- update/edit
- end
- delete
- import

SDP Objects:

- pb (processing block)
- script (processing script definition)
- deployment
- eb (execution block)
- controller (Tango controller device)
- subarray (Tango subarray device)


	pb	script	deployment	eb	other
list	<ul style="list-style-type: none">list all pbslist pbs for a certain date	<ul style="list-style-type: none">list all script definitionslist a script def of a specific kind (batch or realtime)	list all deployments	list all ebs	<ul style="list-style-type: none">if -a -all: list all the contents of the Config DBif -v -values: list keys with values (or just values)if -prefix: list limited to this prefix (for testing purposes)if controller, list the device entry if there is oneif subarray, list all subarray device entries
get/watch	<ul style="list-style-type: none">get the value of a single keyget the values of all pb-related keys for a single pb-id	get the value of a single key	get the value of a single key	get the value of a single key	<i>Note: rules for get and watch are the same</i>
create	<ul style="list-style-type: none">create a pb to run a processing scriptif -eb: add eb parameters for real-time pb	create a key/value pair with prefix of /script	create a deployment of given deployment-id, kind, and parameters	create a key/value pair with prefix of /eb	<i>Not implemented for Tango devices</i>
38					Chapter 5. SDP command-line interface

5.2 Relevant environment variables

Backend-related:

SDP_CONFIG_BACKEND	Database backend (default etcd3)
SDP_CONFIG_HOST	Database host address (default 127.0.0.1)
SDP_CONFIG_PORT	Database port (default 2379)
SDP_CONFIG_PROTOCOL	Database access protocol (default http)
SDP_CONFIG_CERT	Client certificate
SDP_CONFIG_USERNAME	User name
SDP_CONFIG_PASSWORD	User password

When running *ska-sdp edit*:

EDITOR Executable of an existing text editor. Recommended: vi, vim, nano (i.e.  command line-based editors)

5.3 Usage

```
> ska-sdp --help
```

Command line utility for interacting with SKA Science Data Processor (SDP).

Usage:

```
ska-sdp COMMAND [options] [SDP_OBJECT] [<args>...]
ska-sdp COMMAND (-h|--help)
ska-sdp (-h|--help)
```

SDP Objects:

```
pb          Interact with processing blocks
script      Interact with available processing script definitions
deployment  Interact with deployments
eb          Interact with execution blocks
controller  Interact with Tango controller device
subarray    Interact with Tango subarray device
```

Commands:

```
list        List information of object from the Configuration DB
get | watch Print all the information (i.e. value) of a key in the Config DB
create      Create a new, raw key-value pair in the Config DB;
            Run a processing script; Create a deployment
update      Update a raw key value from CLI
edit        Edit a raw key value from text editor
delete      Delete a single key or all keys within a path from the Config DB
end         Stop/Cancel execution block
import      Import processing script definitions from file or URL
```

```
> ska-sdp list --help
```

List keys (and optionally values) within the Configuration Database.

(continues on next page)

(continued from previous page)

Usage:

```
ska-sdp list (-a |--all) [options]
ska-sdp list [options] pb [<date>]
ska-sdp list [options] script [<kind>]
ska-sdp list [options] (deployment|eb|controller|subarray)
ska-sdp list (-h|--help)
```

Arguments:

<date> Date on which the processing block(s) were created. Expected format: `YYYYMMDD`
 ↪ If not provided, all pbs are listed.

<kind> Kind of processing script definition. Batch or realtime.
 ↪ If not provided, all scripts are listed.

Options:

-h, --help Show this screen
 -q, --quiet Cut back on unnecessary output
 -a, --all List the contents of the Config DB, regardless of object type
 -v, --values List all the values belonging to a key in the config db; default: `False`
 ↪ --prefix=<prefix> Path prefix (if other than standard Config paths, e.g. for `testing`)

```
> ska-sdp (get|watch) --help
```

Get/Watch all information of a single key in the Configuration Database.

Usage:

```
ska-sdp (get|watch) [options] <key>
ska-sdp (get|watch) [options] pb <pb-id>
ska-sdp (get|watch) (-h|--help)
```

Arguments:

<key> Key within the Config DB.
 To get the list of all keys:
 ska-sdp list -a

<pb-id> Processing block id to list all entries and their values for.
 Else, use key to get the value of a specific pb.

Options:

-h, --help Show this screen
 -q, --quiet Cut back on unnecessary output

```
> ska-sdp create --help
```

Create SDP objects (deployment, script, eb) in the Configuration Database.
 Create a processing block to run a script.

Usage:

```
ska-sdp create [options] pb <script> [<parameters>] [--eb=<eb-parameters>]
```

(continues on next page)

(continued from previous page)

```
ska-sdp create [options] deployment <item-id> <kind> <parameters>
ska-sdp create [options] (script|eb) <item-id> <value>
ska-sdp create (-h|--help)
```

Arguments:

```
<script>          Script that the processing block will run, in the format:
                   kind:name:version
<parameters>      Optional parameters for a script, with expected format:
                   '{"key1": "value1", "key2": "value2"}'
                   For deployments, expected format:
                   '{"chart": <chart-name>, "values": <dict-of-values>}'
<eb-parameters>   Optional eb parameters for a real-time script
<item-id>          Id of the new deployment, script or eb
<kind>             Kind of the new deployment (currently "helm" only)
```

Options:

```
-h, --help      Show this screen
-q, --quiet     Cut back on unnecessary output
```

Example:

```
ska-sdp create eb eb-test-20210524-000000 '{"test": true}'
Result in the config db:
  key: /eb/eb-test-20210524-000000
  value: {"test": true}
```

Note: You cannot create processing blocks apart from when they are called to run a `script`.

```
> ska-sdp (update|edit) --help
```

Update the value of a single key or processing block state.
Can either update from CLI, or edit via a text editor.

Usage:

```
ska-sdp update [options] (script|eb|deployment) <item-id> <value>
ska-sdp update [options] pb-state <item-id> <value>
ska-sdp update [options] controller <value>
ska-sdp update [options] subarray <item-id> <value>
ska-sdp edit (script|eb|deployment) <item-id>
ska-sdp edit pb-state <item-id>
ska-sdp edit controller
ska-sdp edit subarray <item-id>
ska-sdp (update|edit) (-h|--help)
```

Arguments:

```
<item-id>    id of the script, eb, deployment, processing block or subarray
<value>      Value to update the key/pb state with.
```

Options:

```
-h, --help      Show this screen
-q, --quiet     Cut back on unnecessary output
```

(continues on next page)

(continued from previous page)

Note:

ska-sdp edit needs an environment variable defined:
 EDITOR: Has to match the executable of an existing text editor
 Recommended: vi, vim, nano (i.e. command line-based editors)
 Example: EDITOR=vi ska-sdp edit <key>
 Processing blocks cannot be changed, apart from their state.

Example:

```
ska-sdp edit eb eb-test-20210524-000000
--> key that's edited: /eb/eb-test-20210524-000000
ska-sdp edit script batch:test:0.0.0
--> key that's edited: /script/batch:test:0.0.0
ska-sdp edit pb-state some-pb-id-0000
--> key that's edited: /pb/some-pb-id-0000/state
```

```
> ska-sdp delete --help
```

Delete a key from the Configuration Database.

Usage:

```
ska-sdp delete (-a|--all) [options] (pb|script|eb|deployment|prefix)
ska-sdp delete [options] (pb|eb|deployment) <item-id>
ska-sdp delete [options] script <script>
ska-sdp delete (-h|--help)
```

Arguments:

```
<item-id>  ID of the processing block, or deployment, or execution block
<script>   Script definition to be deleted. Expected format: kind:name:version
prefix     Use this "SDP Object" when deleting with a non-object-specific, user-
↳defined prefix
```

Options:

```
-h, --help          Show this screen
-q, --quiet         Cut back on unnecessary output
--prefix=<prefix>   Path prefix (if other than standard Config paths, e.g. for
↳testing)
```

```
> ska-sdp end --help
```

End execution block in the configuration database.

By default it sets the status to FINISHED. If the --cancel flag is set, it sets the status to CANCELLED.

Usage:

```
ska-sdp end eb <eb-id> [options]
ska-sdp end (-h|--help)
```

Arguments:

```
<eb-id>  ID of execution block to end
```

Options:

(continues on next page)

(continued from previous page)

```
-c, --cancel  Cancel the execution block
-h, --help    Show this screen
-q, --quiet   Cut back on unnecessary output
```

```
> ska-sdp import --help
```

Import processing script definitions into the Configuration Database.

Usage:

```
ska-sdp import scripts [options] <file-or-url>
ska-sdp import (-h|--help)
```

Arguments:

```
<file-or-url>      File or URL to import script definitions from.
```

Options:

```
-h, --help          Show this screen
--sync             Delete scripts not in the input
```

5.4 Example script definitions file

You can also use a script definitions file to import processing scripts into the Config DB. An example script definitions file looks like

```
scripts:
- kind: realtime
  name: test_realtime
  version: 0.2.2
  image: artefact.skao.int/ska-sdp-script-test-realtime:0.2.2
- kind: batch
  name: test_batch
  version: 0.2.2
  image: artefact.skao.int/ska-sdp-script-test-batch:0.2.2
```

Both YAML and JSON files are accepted. After the import, you can check via .. code-block:: bash

```
ska-sdp list script
```

It will output a list of processing scripts that are available to use.

INDICES AND TABLES

- `genindex`
- `modindex`

PYTHON MODULE INDEX

S

- `ska_sdp_config.backend.common`, [23](#)
- `ska_sdp_config.backend.etcd3`, [23](#)
- `ska_sdp_config.backend.etcd3_revolution1`, [27](#)
- `ska_sdp_config.backend.memory`, [32](#)
- `ska_sdp_config.config`, [15](#)
- `ska_sdp_config.entity.deployment`, [23](#)
- `ska_sdp_config.entity.pb`, [22](#)

INDEX

A

`alive_key` (*ska_sdp_config.config.Config* property), 15
`args` (*ska_sdp_config.entity.deployment.Deployment* property), 23

B

`backend` (*ska_sdp_config.config.Config* property), 15

C

`clear_watch()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 30
`client_lease` (*ska_sdp_config.config.Config* property), 15
`close()` (*ska_sdp_config.backend.etcd3.Etcd3Backend* method), 23
`close()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1* method), 27
`close()` (*ska_sdp_config.backend.memory.MemoryBackend* method), 32
`close()` (*ska_sdp_config.config.Config* method), 15
`commit()` (*ska_sdp_config.backend.etcd3.Etcd3Transaction* method), 25
`commit()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 30
`commit()` (*ska_sdp_config.backend.memory.MemoryTransaction* method), 34
`Config` (class in *ska_sdp_config.config*), 15
`ConfigCollision`, 23
`ConfigVanished`, 23
`create()` (*ska_sdp_config.backend.etcd3.Etcd3Backend* method), 23
`create()` (*ska_sdp_config.backend.etcd3.Etcd3Transaction* method), 26
`create()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1* method), 27
`create()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 30
`create()` (*ska_sdp_config.backend.memory.MemoryBackend* method), 32
`create()` (*ska_sdp_config.backend.memory.MemoryTransaction* method), 34
`create_controller()` (*ska_sdp_config.config.Transaction* method), 16
`create_deployment()` (*ska_sdp_config.config.Transaction* method), 16
`create_deployment_state()` (*ska_sdp_config.config.Transaction* method), 16
`create_execution_block()` (*ska_sdp_config.config.Transaction* method), 16
`create_is_alive()` (*ska_sdp_config.config.Transaction* method), 17
`create_processing_block()` (*ska_sdp_config.config.Transaction* method), 17
`create_processing_block_state()` (*ska_sdp_config.config.Transaction* method), 17
`create_script()` (*ska_sdp_config.config.Transaction* method), 17
`create_subarray()` (*ska_sdp_config.config.Transaction* method), 17
`delete()` (*ska_sdp_config.backend.etcd3.Etcd3Backend* method), 24
`delete()` (*ska_sdp_config.backend.etcd3.Etcd3Transaction* method), 26
`delete()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1* method), 27
`delete()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 30
`delete()` (*ska_sdp_config.backend.memory.MemoryBackend* method), 33
`delete()` (*ska_sdp_config.backend.memory.MemoryTransaction* method), 34
`delete_deployment()` (*ska_sdp_config.config.Transaction* method), 17
`delete_execution_block()`

(*ska_sdp_config.config.Transaction* method), 17
 delete_processing_block()
 (*ska_sdp_config.config.Transaction* method), 18
 delete_script() (*ska_sdp_config.config.Transaction* method), 18
 dependencies (*ska_sdp_config.entity.pb.ProcessingBlock* property), 22
 Deployment (class in *ska_sdp_config.entity.deployment*), 23
 depth_of_path() (in module *ska_sdp_config.backend.common*), 23
 dict_to_json() (in module *ska_sdp_config.config*), 22
 dpl_id (*ska_sdp_config.entity.deployment.Deployment* property), 23

E

eb_id (*ska_sdp_config.entity.pb.ProcessingBlock* property), 22
 Etcd3Backend (class in *ska_sdp_config.backend.etcd3*), 23
 Etcd3BackendRevolution1 (class in *ska_sdp_config.backend.etcd3_revolution1*), 27
 Etcd3Transaction (class in *ska_sdp_config.backend.etcd3*), 25
 Etcd3Transaction (class in *ska_sdp_config.backend.etcd3_revolution1*), 29
 Etcd3Watch (class in *ska_sdp_config.backend.etcd3_revolution1*), 31
 Etcd3Watcher (class in *ska_sdp_config.backend.etcd3_revolution1*), 32

G

get() (*ska_sdp_config.backend.etcd3.Etcd3Backend* method), 24
 get() (*ska_sdp_config.backend.etcd3.Etcd3Transaction* method), 26
 get() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1* method), 28
 get() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 30
 get() (*ska_sdp_config.backend.memory.MemoryBackend* method), 33
 get() (*ska_sdp_config.backend.memory.MemoryTransaction* method), 35
 get_controller() (*ska_sdp_config.config.Transaction* method), 18
 get_deployment() (*ska_sdp_config.config.Transaction* method), 18
 get_deployment_state()
 (*ska_sdp_config.config.Transaction* method), 18

get_execution_block()
 (*ska_sdp_config.config.Transaction* method), 18
 get_processing_block()
 (*ska_sdp_config.config.Transaction* method), 18
 get_processing_block_owner()
 (*ska_sdp_config.config.Transaction* method), 18
 get_processing_block_state()
 (*ska_sdp_config.config.Transaction* method), 19
 get_script() (*ska_sdp_config.config.Transaction* method), 19
 get_subarray() (*ska_sdp_config.config.Transaction* method), 19

I

is_alive() (*ska_sdp_config.config.Config* method), 15
 is_alive() (*ska_sdp_config.config.Transaction* method), 19
 is_processing_block_owner()
 (*ska_sdp_config.config.Transaction* method), 19

K

kind (*ska_sdp_config.entity.deployment.Deployment* property), 23

L

lease() (*ska_sdp_config.backend.etcd3.Etcd3Backend* method), 24
 lease() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1* method), 28
 lease() (*ska_sdp_config.backend.memory.MemoryBackend* method), 33
 lease() (*ska_sdp_config.config.Config* method), 15
 list_deployments() (*ska_sdp_config.config.Transaction* method), 19
 list_execution_blocks()
 (*ska_sdp_config.config.Transaction* method), 20
 list_keys() (*ska_sdp_config.backend.etcd3.Etcd3Backend* method), 24
 list_keys() (*ska_sdp_config.backend.etcd3.Etcd3Transaction* method), 26
 list_keys() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1* method), 28
 list_keys() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 30
 list_keys() (*ska_sdp_config.backend.memory.MemoryBackend* method), 33
 list_keys() (*ska_sdp_config.backend.memory.MemoryTransaction* method), 35

list_processing_blocks()
(*ska_sdp_config.config.Transaction* method), 20

list_scripts() (*ska_sdp_config.config.Transaction* method), 20

list_subarrays() (*ska_sdp_config.config.Transaction* method), 20

loop() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* property), 27
(method), 30

loop() (*ska_sdp_config.backend.memory.MemoryTransaction* method), 35

loop() (*ska_sdp_config.config.Transaction* method), 20

M

MemoryBackend (class
ska_sdp_config.backend.memory), 32

MemoryTransaction (class
ska_sdp_config.backend.memory), 34

MemoryWatcher (class
ska_sdp_config.backend.memory), 35

module

- ska_sdp_config.backend.common*, 23
- ska_sdp_config.backend.etcd3*, 23
- ska_sdp_config.backend.etcd3_revolution1*, 27
- ska_sdp_config.backend.memory*, 32
- ska_sdp_config.config*, 15
- ska_sdp_config.entity.deployment*, 23
- ska_sdp_config.entity.pb*, 22

N

new_execution_block_id()
(*ska_sdp_config.config.Transaction* method), 20

new_processing_block_id()
(*ska_sdp_config.config.Transaction* method), 20

O

on_commit() (*ska_sdp_config.backend.etcd3.Etcd3Transaction* method), 26

on_commit() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 31

P

parameters (*ska_sdp_config.entity.pb.ProcessingBlock* property), 22

pb_id (*ska_sdp_config.entity.pb.ProcessingBlock* property), 22

ProcessingBlock (class in *ska_sdp_config.entity.pb*), 22

R

raw (*ska_sdp_config.config.Transaction* property), 21

reset() (*ska_sdp_config.backend.etcd3.Etcd3Transaction* method), 26

reset() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 31

reset() (*ska_sdp_config.backend.memory.MemoryTransaction* method), 35

revision(*ska_sdp_config.backend.etcd3.Etcd3Transaction* property), 27

revision(*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* property), 31

S

script (*ska_sdp_config.entity.pb.ProcessingBlock* property), 22

in set_alive() (*ska_sdp_config.config.Config* method), 15

ska_sdp_config.backend.common

in module, 23

ska_sdp_config.backend.etcd3

in module, 23

ska_sdp_config.backend.etcd3_revolution1

module, 27

ska_sdp_config.backend.memory

module, 32

ska_sdp_config.config

module, 15

ska_sdp_config.entity.deployment

module, 23

ska_sdp_config.entity.pb

module, 22

start() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Watch* method), 31

stop() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Watch* method), 32

T

take_processing_block()
(*ska_sdp_config.config.Transaction* method), 21

to_dict() (*ska_sdp_config.entity.deployment.Deployment* method), 23

to_dict() (*ska_sdp_config.entity.pb.ProcessingBlock* method), 22

Transaction (class in *ska_sdp_config.config*), 16

trigger() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Watcher* method), 32

trigger_loop() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction* method), 31

txn() (*ska_sdp_config.backend.etcd3.Etcd3Backend* method), 25

txn() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1* method), 28

txn() (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Watcher* method), 32

`txn()` (*ska_sdp_config.backend.memory.MemoryBackend*
method), 33
`txn()` (*ska_sdp_config.backend.memory.MemoryWatcher*
method), 35
`txn()` (*ska_sdp_config.config.Config* *method*), 16

U

`update()` (*ska_sdp_config.backend.etcd3.Etcd3Backend*
method), 25
`update()` (*ska_sdp_config.backend.etcd3.Etcd3Transaction*
method), 27
`update()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1*
method), 29
`update()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction*
method), 31
`update()` (*ska_sdp_config.backend.memory.MemoryBackend*
method), 34
`update()` (*ska_sdp_config.backend.memory.MemoryTransaction*
method), 35
`update_controller()`
(*ska_sdp_config.config.Transaction* *method*),
21
`update_deployment_state()`
(*ska_sdp_config.config.Transaction* *method*),
21
`update_execution_block()`
(*ska_sdp_config.config.Transaction* *method*),
21
`update_processing_block()`
(*ska_sdp_config.config.Transaction* *method*),
21
`update_processing_block_state()`
(*ska_sdp_config.config.Transaction* *method*),
21
`update_script()` (*ska_sdp_config.config.Transaction*
method), 21
`update_subarray()` (*ska_sdp_config.config.Transaction*
method), 22

W

`watch()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1*
method), 29
`watch()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3Transaction*
method), 31
`watcher()` (*ska_sdp_config.backend.etcd3.Etcd3Backend*
method), 25
`watcher()` (*ska_sdp_config.backend.etcd3_revolution1.Etcd3BackendRevolution1*
method), 29
`watcher()` (*ska_sdp_config.backend.memory.MemoryBackend*
method), 34
`watcher()` (*ska_sdp_config.config.Config* *method*), 16