# Native Programming Interface

## Building against the DAOS library

To build an application or I/O middleware against the native DAOS API, include the daos.h header file in your program and link with -Ldaos. Examples are available under src/tests.

## DAOS API Reference

libdaos is written in C and uses Doxygen comments that are added to C header files. The Doxygen documentation is available [here](https://daos-stack.github.io/html/).

## Python Bindings

A python module called [PyDAOS](https://github.com/daos-stack/daos/blob/master/src/client/pydaos) provides the DAOS API to python users.

### pydaos

[pydaos](https://github.com/daos-stack/daos/blob/master/src/client/pydaos/pydaos_core.py) provides a native DAOS python interface exported by a C module. It integrates the DAOS key-value store API with python dictionaries. Only strings are supported for both the key and value for now.

Key-value pairs can be inserted/looked up one at a time (see put/get) or in bulk (see bput/bget), taking a python dict as an input. The bulk operations are issued in parallel (up to 16 operations in flight) to maximize the operation rate.

Key-value pairs are deleted via the put/bput operations by setting the value to either None or the empty string. Once deleted, the key won’t be reported during iteration. It also supports the del operation via ‘del dkv.key’. The DAOS KV objects behave like a python dictionary and support:

* ‘dkv[key]’ which invokes ‘dkv.get(key)’
* ‘dkv[key] = val’ which invokes ‘dkv.put(key, val)’
* ‘for key in dkv:’ allows for walking through the key space via the support of python iterators
* ‘if key is in dkv:’ allows testing whether a given key is present in the DAOS KV store.
* ‘len(dkv)’ returns the number of key-value pairs.
* ‘bool(dkv)’ reports ‘False’ if there are no key-value pairs in the DAOS KV and ‘True’ otherwise.

Python iterators are supported, which means that “for key in kvobj:” will allow you to walk through the key space. For each method, a PyDError exception is raised with a proper DAOS error code (in string format) if the operation cannot be completed.

Both Python 2.7 and 3.x is supported.

### pydaos.raw

The pydaos.raw submodule provides access to DAOS API functionality via Ctypes and was developed with an emphasis on test use cases. While the majority of unit tests are written in C, higher-level tests are written primarily using the Python API. Interfaces are provided for accessing DAOS management and DAOS API functionality from Python. This higher level interface allows a faster turnaround time on implementing test cases for DAOS.

#### Layout

The Python API is split into several files based on functionality:

* The Python object API: [daos\_api.py](https://github.com/daos-stack/daos/tree/master/src/client/pydaos/raw/daos_api.py).
* The mapping of C structures to Python classes [daos\_cref.py](https://github.com/daos-stack/daos/tree/master/src/client/pydaos/raw/daos_cref.py)

High-level abstraction classes exist to manipulate DAOS storage:

class DaosPool(object)  
class DaosContainer(object)  
class DaosObj(object)  
class IORequest(object)

DaosPool is a Python class representing a DAOS pool. All pool-related functionality is exposed from this class. Operations such as creating/destroying a pool, connecting to a pool, and adding a target to a storage pool are supported.

DaosContainer is a Python class representing a DAOS container. As with the DaosPool class, all container-related functionality is exposed here. This class also exposes abstracted wrapper functions for the flow of creating and committing an object to a DAOS container.

DaosObj is a Python class representing a DAOS object. Functionality such as creating/deleting objects in a container, ‘punching’ objects (delete an object from the specified transaction only), and object query.

IORequest is a Python class representing a read or write request against a DAOS object.

Several classes exist for management purposes as well:

class DaosContext(object)  
class DaosLog  
class DaosApiError(Exception)

DaosContext is a wrapper around the DAOS libraries. It is initialized with the path where DAOS libraries can be found.

DaosLog exposes functionality to write messages to the DAOS client log.

DaosApiError is a custom exception class raised by the API internally in the event of a failed DAOS action.

Most functions exposed in the DAOS C API support both synchronous and asynchronous execution, and the Python API exposes this same functionality. Each API takes an input event. DaosEvent is the Python representation of this event. If the input event is NULL, the call is synchronous. If an event is supplied, the function will return immediately after submitting API requests to the underlying stack, and the user can poll and query the event for completion.

#### Ctypes

Ctypes is a built-in Python module for interfacing Python with existing libraries written in C/C++. The Python API is built as an object-oriented wrapper around the DAOS libraries utilizing ctypes.

Ctypes documentation can be found here <https://docs.python.org/3/library/ctypes.html>

The following demonstrates a simplified example of creating a Python wrapper for the C function daos\_pool\_tgt\_exclude\_out, with each input parameter to the C function being cast via ctypes. This also demonstrates struct representation via ctypes:

// daos\_exclude.c  
  
#include <stdio.h>  
  
int  
daos\_pool\_tgt\_exclude\_out(const uuid\_t uuid, const char \*grp,  
 const d\_rank\_list\_t \*svc, struct d\_tgt\_list \*tgts,  
 daos\_event\_t \*ev);

All input parameters must be represented via ctypes. If a struct is required as an input parameter, a corresponding Python class can be created. For struct d\_tgt\_list:

struct d\_tgt\_list {  
 d\_rank\_t \*tl\_ranks;  
 int32\_t \*tl\_tgts;  
 uint32\_t tl\_nr;  
};

class DTgtList(ctypes.Structure):  
 \_fields\_ = [("tl\_ranks", ctypes.POINTER(ctypes.c\_uint32)),  
 ("tl\_tgts", ctypes.POINTER(ctypes.c\_int32)),  
 ("tl\_nr", ctypes.c\_uint32)]

The shared object containing daos\_pool\_tgt\_exclude\_out can then be imported and the function called directly:

# api.py  
  
import ctypes  
import uuid  
import conversion # utility library to convert C <---> Python UUIDs  
  
# init python variables  
p\_uuid = str(uuid.uuid4())  
p\_tgts = 2  
p\_ranks = DaosPool.\_\_pylist\_to\_array([2])  
  
# cast python variables via ctypes as necessary  
c\_uuid = str\_to\_c\_uuid(p\_uuid)  
c\_grp = ctypes.create\_string\_buffer(b"daos\_group\_name")  
c\_svc = ctypes.POINTER(2) # ensure pointers are cast/passed as such  
c\_tgt\_list = ctypes.POINTER(DTgtList(p\_ranks, p\_tgts, 2))) # again, DTgtList must be passed as pointer  
  
# load the shared object  
my\_lib = ctypes.CDLL('/full/path/to/daos\_exclude.so')  
  
# now call it  
my\_lib.daos\_pool\_tgt\_exclude\_out(c\_uuid, c\_grp, c\_svc, c\_tgt\_list, None)

#### Error Handling

The API was designed using the EAFP (Easier to Ask Forgiveness than get Permission) idiom. A given function will raise a custom exception on error state, DaosApiError. A user of the API is expected to catch and handle this exception as needed:

# catch and log  
try:  
 daos\_some\_action()  
except DaosApiError as e:  
 self.d\_log.ERROR("My DAOS action encountered an error!")

#### Logging

The Python DAOS API exposes functionality to log messages to the DAOS client log. Messages can be logged as INFO, DEBUG, WARN, or ERR log levels. The DAOS log object must be initialized with the environmental context in which to run:

from pydaos.raw import DaosLog  
  
self.d\_log = DaosLog(self.context)  
  
self.d\_log.INFO("FYI")  
self.d\_log.DEBUG("Debugging code")  
self.d\_log.WARNING("Be aware, may be issues")  
self.d\_log.ERROR("Something went very wrong")

## Go Bindings

API bindings for Go[[1]](#footnote-37) are also available.

1. https://godoc.org/github.com/daos-stack/go-daos/pkg/daos [↑](#footnote-ref-37)