DRMAA Python Documentation

Release 0.7.1

Dan Blanchard, David Ressman, Enrico Sirola

November 27, 2013

Contents

Contents:

CHAPTER 1

Distributed Resource Management Application API

This guide is a tutorial for getting started programming with DRMAA. It is basically a one to one translation of the original in C for Grid Engine. It assumes that you already know what DRMAA is and that you have drmaa-python installed. If not, have a look at Installing. The following code segments are also included in the repository.

1.1 Starting and Stopping a Session

The following code segments (example1.py and example1.1.py) shows the most basic DRMAA python binding program.

```python
#!/usr/bin/env python

import drmaa

def main():
   """Create a drmaa session and exit"""
   with drmaa.Session() as s:
      print('A session was started successfully')

if __name__=='__main__':
   main()
```

The first thing to notice is that every call to a DRMAA function will return an error code. In this tutorial, we ignore all error codes.

Now let's look at the functions being called. First, on line 7, we initialise a Session object by calling DRMAA.Session(). The Session is automatically initialized via initialize(), and it creates a session and starts an event client listener thread. The session is used for organizing jobs submitted through DRMAA, and the thread is used to receive updates from the queue master about the state of jobs and the system in general. Once initialize() has been called successfully, it is the responsibility of the calling application to also call exit() before terminating. If an application does not call exit() before terminating, session state may be left behind in the user's home directory, and the queue master may be left with a dead event client handle, which can decrease queue master performance.

At the end of our program, on line 9, we call exit(). exit() cleans up the session and stops the event client listener thread. Most other DRMAA functions must be called before exit(). Some functions, like getContact(), can be called after exit(), but these functions only provide general information. Any function that does work, such as runJob() or wait() must be called before exit() is called. If such a function is called after exit() is called, it will return an error.

```python
#!/usr/bin/env python

import drmaa

def main():
    """Create a session, show that each session has an id,
    use session id to disconnect, then reconnect. Then exit"""
    s = drmaa.Session()
    s.initialize()
    print('A session was started successfully')
    response = s.contact
    print('session contact returns: %s' % response)
    s.exit()
    print('Exited from session')

    s.initialize(response)
    print('Session was restarted successfullly')
    s.exit()


if __name__=='__main__':
    main()
```

This example is very similar to Example 1. The difference is that it uses the Grid Engine feature of reconnectable sessions. The DRMAA concept of a session is translated into a session tag in the Grid Engine job structure. That means that every job knows to which session it belongs. With reconnectable sessions, it's possible to initialize the DRMAA library to a previous session, allowing the library access to that session's job list. The only limitation, though, is that jobs which end between the calls to exit() and init() will be lost, as the reconnecting session will no longer see these jobs, and so won't know about them.

Through line 9, this example is very similar to Example 1. On line 10, however, we use the contact attribute to get the contact information for this session. On line 12 we then exit the session. On line 15, we use the stored contact information to reconnect to the previous session. Had we submitted jobs before calling exit(), those jobs would now be available again for operations such as wait() and synchronize(). Finally, on line 17 we exit the session a second time.

1.2 Running a Job

The following code segment (example2.py and example2.1.py) shows how to use the DRMAA python binding to submit a job to Grid Engine. It submits a small shell script (sleeper.sh) which takes two arguments:

```bash
#!/bin/bash
echo "Hello world, the answer is $1"
sleep 3s
echo "$2 Bye world!"
```

```python
#!/usr/bin/env python

import drmaa
import os

def main():
   """Submit a job.
```

```
    Note, need file called sleeper.sh in current directory.
    """
    with drmaa.Session() as s:
        print('Creating job template')
        jt = s.createJobTemplate()
        jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
        jt.args = ['42', 'Simon says:']
        jt.joinFiles=True

        jobid = s.runJob(jt)
        print('Your job has been submitted with id %s' % jobid)

        print('Cleaning up')
        s.deleteJobTemplate(jt)

if __name__=='__main__':
    main()
```

The beginning and end of this program are the same as the previous one. What's different is in lines 13-23. On line 13 we ask DRMAA to allocate a job template for us. A job template is a structure used to store information about a job to be submitted. The same template can be reused for multiple calls to runJob() or runBulkJob().

On line 14 we set the REMOTE_COMMAND attribute. This attribute tells DRMAA where to find the program we want to run. Its value is the path to the executable. The path be be either relative or absolute. If relative, it is relative to the WD attribute, which if not set defaults to the user's home directory. For more information on DRMAA attributes, please see the attributes man page. Note that for this program to work, the script "sleeper.sh" must be in the current directory.

On line 15 we set the V_ARGV attribute. This attribute tells DRMAA what arguments to pass to the executable. For more information on DRMAA attributes, please see the attributes man page.

On line 18 we submit the job with runJob(). DRMAA will place the id assigned to the job into the character array we passed to runJob(). The job is now running as though submitted by qsub or bsub. At this point calling exit() and/or terminating the program will have no effect on the job.

To clean things up, we delete the job template on line 22. This frees the memory DRMAA set aside for the job template, but has no effect on submitted jobs. Finally, on line 23, we call exit().

If instead of a single job we had wanted to submit an array job, we could have replaced the else on line 18 and 19 with the following:

```
jobid = s.runBulkJobs(jt, 1, 30, 2)
print('Your job has submitted with id %s' % jobid)
```

This code segment submits an array job with 15 tasks numbered 1, 3, 5, 7, etc. An important difference to note is that runBulkJobs() returns the job ids in an array. On line 19, we print all the job ids.

1.3 Waiting for a Job

Now we're going to extend our example to include waiting for a job to finish (example3.py, example3.1.py and example3.2.py).

```
#!/usr/bin/env python

import drmaa
import os
```

```
def main():
    """Submit a job and wait for it to finish.
    Note, need file called sleeper.sh in home directory.
    """
    with drmaa.Session() as s:
        print('Creating job template')
        jt = s.createJobTemplate()
        jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
        jt.args = ['42', 'Simon says:']
        jt.joinFiles = True

        jobid = s.runJob(jt)
        print('Your job has been submitted with id %s' % jobid)

        retval = s.wait(jobid, drmaa.Session.TIMEOUT_WAIT_FOREVER)
        print('Job: {0} finished with status {1}'.format(retval.jobId, retval.hasExited))

        print('Cleaning up')
        s.deleteJobTemplate(jt)

if __name__=='__main__':
    main()
```

This example is very similar to Example 2 except for line 21. On line 21 we call wait() to wait for the job to end. We have to give wait() both the id of the job for which we want to wait and a place to write the id of the job for which we actually waited because the job id we pass in could be JOB_IDS_SESSION_ANY, in which case wait() must have a way of tell us which job is the one that made it return. We also have to pass to wait() how long we are willing to wait for the job to finish. This could be a number of seconds, or it could be either TIMEOUT_WAIT_FOREVER or TIMEOUT_NO_WAIT. Lastly, we collect the exit status. Assuming the wait worked, we query the job's exit status on line 22.

An alternative to wait() when working with multiple jobs, such as jobs submitted by runBulkJobs() or multiple calls to runJob() is synchronize(). synchronize() waits for a set of jobs to finish. To use synchronize(), we could replace lines 18-22 with the following:

```
joblist = s.runBulkJobs(jt, 1, 30, 2)
print('Your job has been submitted with id %s' % joblist)

s.synchronize(joblist, drmaa.Session.TIMEOUT_WAIT_FOREVER, True)
```

Line 18 now call runBulkJobs() so that we have several jobs for which to wait. On line 21, instead of calling wait(), we call synchronize(). synchronize() takes only three interesting parameters. The first is the list of ids for which to wait. This list must be a NULL-terminated array of strings. If the special id, JOB_IDS_SESSION_ALL, appears in the array, synchronize() will wait for all jobs submitted via DRMAA during this session, i.e. since initialize() was called. The second is how long to wait for all the jobs in the list to finish. This is the same as the timeout parameter for wait(). The third is whether this call to synchronize() should clean up after the job. After a job completes, it leaves behind accounting information, such as exist status and usage, until either wait() or synchronize() with dispose set to true is called. It is the responsibility of the application to make sure one of these two functions is called for every job. Not doing so creates a memory leak. Note that calling synchronize() with dispose set to true flushes all accounting information for all jobs in the list. If you want to use synchronize() and still recover the accounting information, set dispose to false and call wait() for each job. To do this in Example 3, we would replace lines 18–22 with the following:

```
joblist = s.runBulkJobs(jt, 1, 30, 2)
print('Your job has been submitted with id %s' % joblist)

s.synchronize(joblist, drmaa.Session.TIMEOUT_WAIT_FOREVER, False)
for curjob in joblist:
    print('Collecting job ' + curjob)
    retval = s.wait(curjob, drmaa.Session.TIMEOUT_WAIT_FOREVER)
    print('Job: {0} finished with status {1}'.format(retval.jobId,
                                                      retval.hasExited))
```

What's different is that on line 22 we set dispose to false, and then on lines 23-26 we wait once for each job, printing the exit status and usage information as we did in Example 3.

We pass joblist to synchronise to wait for each job specifically. Otherwise, the wait() could end up waiting for a job submitted after the call to synchronize().

1.4 Controlling a Job

Now let's look at an example of how to control a job from DRMAA (example4.py):

```
#!/usr/bin/env python

import drmaa
import os

def main():
    """Submit a job, then kill it.
    Note, need file called sleeper.sh in home directory.
    """
    with drmaa.Session() as s:
        print('Creating job template')
        jt = s.createJobTemplate()
        jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
        jt.args = ['42', 'Simon says:']
        jt.joinFiles = True

        jobid = s.runJob(jt)
        print('Your job has been submitted with id %s' % jobid)
        # options are: SUSPEND, RESUME, HOLD, RELEASE, TERMINATE
        s.control(jobid, drmaa.JobControlAction.TERMINATE)

        print('Cleaning up')
        s.deleteJobTemplate(jt)

if __name__=='__main__':
    main()
```

This example is very similar to Example 2 except for line 21. On line 21 we use control() to delete the job we just submitted. Aside from deleting the job, we could have also used control() to suspend, resume, hold, or release it. For more information, see the control man page.

Note that control() can be used to control jobs not submitted through DRMAA. Any valid SGE job id could be passed to control() as the id of the job to delete.

1.5 Getting Job Status

Here's an example of using DRMAA to query the status of a job (example5.py):

```
#!/usr/bin/env python

import drmaa
```

```python
import time
import os

def main():
    """Submit a job, and check its progress.
    Note, need file called sleeper.sh in home directory.
    """
    with drmaa.Session() as s:
        print('Creating job template')
        jt = s.createJobTemplate()
        jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
        jt.args = ['42', 'Simon says:']
        jt.joinFiles=True

        jobid = s.runJob(jt)
        print('Your job has been submitted with id %s' % jobid)

        # Who needs a case statement when you have dictionaries?
        decodestatus = {drmaa.JobState.UNDETERMINED: 'process status cannot be determined',
                        drmaa.JobState.QUEUED_ACTIVE: 'job is queued and active',
                        drmaa.JobState.SYSTEM_ON_HOLD: 'job is queued and in system hold',
                        drmaa.JobState.USER_ON_HOLD: 'job is queued and in user hold',
                        drmaa.JobState.USER_SYSTEM_ON_HOLD: 'job is queued and in user and system hol
                        drmaa.JobState.RUNNING: 'job is running',
                        drmaa.JobState.SYSTEM_SUSPENDED: 'job is system suspended',
                        drmaa.JobState.USER_SUSPENDED: 'job is user suspended',
                        drmaa.JobState.DONE: 'job finished normally',
                        drmaa.JobState.FAILED: 'job finished, but failed'}

        for ix in range(10):
            print('Checking %s of 10 times' % ix)
            print decodestatus(s.jobStatus(jobid))
            time.sleep(5)

        print('Cleaning up')
        s.deleteJobTemplate(jt)

if __name__=='__main__':
    main()
```

Again, this example is very similar to Example 2, this time with the exception of lines 22-40. On line 38, we use jobStatus() to get the status of the job. Line 43 determine what the job status is and report it.

1.6 Getting DRM information

Lastly, let's look at how to query the DRMAA library for information about the DRMS and the DRMAA implementation itself (example6.py):

```python
#!/usr/bin/env python

import drmaa

def main():
    """Query the system."""
    with drmaa.Session() as s:
        print('A DRMAA object was created')
        print('Supported contact strings: %s' % s.contact)
        print('Supported DRM systems: %s' % s.drmsInfo)
        print('Supported DRMAA implementations: %s' % s.drmaaImplementation)
        print('Version %s' % s.version)

        print('Exiting')

if __name__=='__main__':
    main()
```

On line 9, we get the contact string list. This is the list of contact strings that will be understood by this DRMAA instance. Normally on of these strings is used to select to which DRM this DRMAA instance should be bound. On line 10, we get the list of supported DRM systems. On line 11, we get the list of supported DRMAA implementations. On line 12, we get the version number of the DRMAA C binding specification supported by this DRMAA implementation. Finally, on line 15, we end the session with exit().

drmaa Package

A python package for DRM job submission and control.

This package is an implementation of the DRMAA 1.0 Python language binding specification (http://www.ogf.org/documents/GFD.143.pdf). See http://drmaa-python.googlecode.com for package info and download.

author Enrico Sirola (enrico.sirola@statpro.com)

author Dan Blanchard (dblanchard@ets.org)

class `drmaa.JobInfo`

Bases: `tuple`

JobInfo(jobId, hasExited, hasSignal, terminatedSignal, hasCoreDump, wasAborted, exitStatus, resourceUsage)

`exitStatus`
Alias for field number 6

`hasCoreDump`
Alias for field number 4

`hasExited`
Alias for field number 1

`hasSignal`
Alias for field number 2

`jobId`
Alias for field number 0

`resourceUsage`
Alias for field number 7

`terminatedSignal`
Alias for field number 3

`wasAborted`
Alias for field number 5

class `drmaa.JobTemplate`(***kwargs*)

Bases: `object`

A job to be submitted to the DRM.

`HOME_DIRECTORY` = u'$drmaa_hd_ph$'

`PARAMETRIC_INDEX` = u'$drmaa_incr_ph$'

`WORKING_DIRECTORY` = u'$drmaa_wd_ph$'

`attributeNames`

The list of supported DRMAA scalar attribute names.

This is apparently useless now, and should probably substituted by the list of attribute names of the JobTemplate instances.

`blockEmail` = False

`deadlineTime` = u''

`delete`()

Deallocate the underlying DRMAA job template.

`errorPath` = u''

`inputPath` = u''

`jobCategory` = u''

`jobName` = u''

`jobSubmissionState` = u''

`joinFiles` = False

`nativeSpecification` = u''

`outputPath` = u''

`remoteCommand` = u''

`startTime` = u''

`transferFiles` = u''

`workingDirectory` = u''

class `drmaa.`**`Session`**(*contactString=None*)

Bases: `object`

The DRMAA Session.

This class is the entry point for communicating with the DRM system

`JOB_IDS_SESSION_ALL` = 'DRMAA_JOB_IDS_SESSION_ALL'

`JOB_IDS_SESSION_ANY` = 'DRMAA_JOB_IDS_SESSION_ANY'

`TIMEOUT_NO_WAIT` = 0

`TIMEOUT_WAIT_FOREVER` = -1

`contact` = u''

static `control`(*jobId*, *operation*)

Used to hold, release, suspend, resume, or kill the job identified by jobId.

Parameters

jobId [string] if jobId is *Session.JOB_IDS_SESSION_ALL* then this routine acts on all jobs submitted during this DRMAA session up to the moment control() is called. The legal values for action and their meanings are

operation [string]

possible values are:

JobControlAction.SUSPEND stop the job

JobControlAction.RESUME (re)start the job

JobControlAction.HOLD put the job on-hold

JobControlAction.RELEASE release the hold on the job

JobControlAction.TERMINATE kill the job

To avoid thread races in multithreaded applications, the DRMAA implementation user should explicitly synchronize this call with any other job submission calls or control calls that may change the number of remote jobs.

This method returns once the action has been acknowledged by the DRM system, but does not necessarily wait until the action has been completed. Some DRMAA implementations may allow this method to be used to control jobs submitted external to the DRMAA session, such as jobs submitted by other DRMAA session in other DRMAA implementations or jobs submitted via native utilities.

static `createJobTemplate`()

Allocates a new job template.

The job template is used to set the environment for jobs to be submitted. Once the job template has been created, it should also be deleted (via deleteJobTemplate()) when no longer needed. Failure to do so may result in a memory leak.

static `deleteJobTemplate`(*jobTemplate*)

Deallocate a job template.

Parameters

jobTemplate [JobTemplate] the job temptare to be deleted

This routine has no effect on running jobs.

`drmaaImplementation` = u'‘

`drmsInfo` = u'‘

static `exit`()

Used to disengage from DRM.

This routine ends the current DRMAA session but doesn't affect any jobs (e.g., queued and running jobs remain queued and running). exit() should be called only once, by only one of the threads. Additional calls to exit() beyond the first will throw a NoActiveSessionException.

static `initialize`(*contactString=None*)

Used to initialize a DRMAA session for use.

Parameters

contactString [string or None] implementation-dependent string that may be used to specify which DRM system to use

This method must be called before any other DRMAA calls. If contactString is None, the default DRM system is used, provided there is only one DRMAA implementation available. If there is more than one DRMAA implementation available, initialize() throws a NoDefaultContactStringSelectedException. initialize() should be called only once, by only one of the threads. The main thread is recommended. A call to initialize() by another thread or additional calls to initialize() by the same thread with throw a SessionAlreadyActiveException.

static `jobStatus` (*jobId*)
returns the program status of the job identified by jobId.

The possible values returned from this method are:

- *JobState.UNDETERMINED*: process status cannot be determined,
- *JobState.QUEUED_ACTIVE*: job is queued and active,
- *JobState.SYSTEM_ON_HOLD*: job is queued and in system hold,
- *JobState.USER_ON_HOLD*: job is queued and in user hold,
- ***JobState.USER_SYSTEM_ON_HOLD*: job is queued and in user and** system hold,
- *JobState.RUNNING*: job is running,
- *JobState.SYSTEM_SUSPENDED*: job is system suspended,
- *JobState.USER_SUSPENDED*: job is user suspended,
- *JobState.DONE*: job finished normally, and
- *JobState.FAILED*: job finished, but failed.

The DRMAA implementation should always get the status of the job from the DRM system unless the status has already been determined to be FAILED or DONE and the status has been successfully cached. Terminated jobs return a FAILED status.

static `runBulkJobs` (*jobTemplate*, *beginIndex*, *endIndex*, *step*)
Submit a set of parametric jobs, each with attributes defined in the job template.

Parameters

jobTemplate [JobTemplate] the template representng jobs to be run

beginIndex [int] index of the first job

endIndex [int] index of the last job

step [int] the step between job ids

The returned job identifiers are Strings identical to those returned from the underlying DRM system. The JobTemplate class defines a *JobTemplate.PARAMETRIC_INDEX* placeholder for use in specifying paths. This placeholder is used to represent the individual identifiers of the tasks submitted through this method.

static `runJob` (*jobTemplate*)
Submit a job with attributes defined in the job template.

Parameters

jobTemplate [JobTemplate] the template representing the job to be run

The returned job identifier is a String identical to that returned from the underlying DRM system.

static `synchronize` (*jobIds*, *timeout=-1*, *dispose=False*)
Waits until all jobs specified by jobList have finished execution.

Parameters

jobIds If jobIds contains *Session.JOB_IDS_SESSION_ALL*, then this method waits for all jobs submitted during this DRMAA session up to the moment synchronize() is called

timeout [int] maximum time (in seconds) to be waited for the completion of a job.

The value *Session.TIMEOUT_WAIT_FOREVER* may be specified to wait indefinitely for a result. The value *Session.TIMEOUT_NO_WAIT* may be specified to return immediately if no result is available.

dispose [bool] specifies how to treat the reaping of the remote job's internal data record, which includes a record of the job's consumption of system resources during its execution and other statistical information. If set to True, the DRM will dispose of the job's data record at the end of the synchroniize() call. If set to False, the data record will be left for future access via the wait() method.

To avoid thread race conditions in multithreaded applications, the DRMAA implementation user should explicitly synchronize this call with any other job submission calls or control calls that may change the number of remote jobs.

If the call exits before the timeout has elapsed, all the jobs have been waited on or there was an interrupt. If the invocation exits on timeout, an ExitTimeoutException is thrown. The caller should check system time before and after this call in order to be sure of how much time has passed.

`version` **= Version(major=10L, minor=10L)**

static `wait` (*jobId*, *timeout=-1*)

Wait for a job with jobId to finish execution or fail.

Parameters

jobId [str] The job id to wait completion for.

If the special string, *Session.JOB_IDS_SESSION_ANY*, is provided as the jobId, this routine will wait for any job from the session

timeout [float] The timeout value is used to specify the desired behavior when a result is not immediately available.

The value *Session.TIMEOUT_WAIT_FOREVER* may be specified to wait indefinitely for a result. The value *Session.TIMEOUT_NO_WAIT* may be specified to return immediately if no result is available. Alternatively, a number of seconds may be specified to indicate how long to wait for a result to become available

This routine is modeled on the wait3 POSIX routine. If the call exits before timeout, either the job has been waited on successfully or there was an interrupt. If the invocation exits on timeout, an *ExitTimeoutException* is thrown. The caller should check system time before and after this call in order to be sure how much time has passed. The routine reaps job data records on a successful call, so any subsequent calls to wait() will fail, throwing an *InvalidJobException*, meaning that the job's data record has been already reaped. This exception is the same as if the job were unknown. (The only case where wait() can be successfully called on a single job more than once is when the previous call to wait() timed out before the job finished.)

exception `drmaa.`**`AlreadyActiveSessionException`**

Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`AuthorizationException`**

Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`ConflictingAttributeValuesException`**

Bases: `drmaa.errors.DrmaaException`, `exceptions.AttributeError`

exception `drmaa.`**`DefaultContactStringException`**

Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`DeniedByDrmException`**

Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`DrmCommunicationException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`DrmsExitException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`DrmsInitException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`ExitTimeoutException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`HoldInconsistentStateException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`IllegalStateException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`InternalException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`InvalidAttributeFormatException`**
Bases: `drmaa.errors.DrmaaException`, `exceptions.AttributeError`

exception `drmaa.`**`InvalidContactStringException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`InvalidJobException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`InvalidJobTemplateException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`NoActiveSessionException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`NoDefaultContactStringSelectedException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`ReleaseInconsistentStateException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`ResumeInconsistentStateException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`SuspendInconsistentStateException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`TryLaterException`**
Bases: `drmaa.errors.DrmaaException`

exception `drmaa.`**`UnsupportedAttributeException`**
Bases: `drmaa.errors.DrmaaException`, `exceptions.AttributeError`

exception `drmaa.`**`InvalidArgumentException`**
Bases: `drmaa.errors.DrmaaException`, `exceptions.AttributeError`

exception `drmaa.`**`InvalidAttributeValueException`**
Bases: `drmaa.errors.DrmaaException`, `exceptions.AttributeError`

exception `drmaa.`**`OutOfMemoryException`**
Bases: `drmaa.errors.DrmaaException`, `exceptions.MemoryError`

`drmaa.`**`control_action_to_string`**(*code*) `drmaa.`**`job_state`**`(`*code*`)`

class `drmaa.`**`JobControlAction`**

Bases: `object`

`HOLD` **= u'hold'**

`RELEASE` **= u'release'**

`RESUME` **= u'resume'**

`SUSPEND` **= u'suspend'**

`TERMINATE` **= u'terminate'**

class `drmaa.`**`JobState`**

Bases: `object`

`DONE` **= u'done'**

`FAILED` **= u'failed'**

`QUEUED_ACTIVE` **= u'queued_active'**

`RUNNING` **= u'running'**

`SYSTEM_ON_HOLD` **= u'system_on_hold'**

`SYSTEM_SUSPENDED` **= u'system_suspended'**

`UNDETERMINED` **= u'undetermined'**

`USER_ON_HOLD` **= u'user_on_hold'**

`USER_SUSPENDED` **= u'user_suspended'**

`USER_SYSTEM_ON_HOLD` **= u'user_system_on_hold'**

`USER_SYSTEM_SUSPENDED` **= u'user_system_suspended'**

class `drmaa.`**`JobSubmissionState`**

Bases: `object`

`ACTIVE_STATE` **= u'drmaa_active'**

`HOLD_STATE` **= u'drmaa_hold'**

`drmaa.`**`status_to_string`**`(`*status*`)`

`drmaa.`**`string_to_control_action`**`(`*operation*`)`

`drmaa.`**`submission_state`**`(`*code*`)`

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index