Job Author Python API

tractor.api.author is a Python API for building Tractor jobs.

This module is installed with the Python interpreter that ships with Tractor, rmanpy. It may be used with other Python interpreters, but will require site-specific configuration to locate or install the required Tractor Python modules.

The following examples assume that the query module has been imported as follows:

```
>>> import tractor.api.author as author
```

Overview

A job is built using the Job, Task, Instance, and Command classes. It can then be spooled using the Job.spool() method or displayed using the Job.asTcl() method. Here is a simple example:

```
>>> job = author.Job(title="a one-task render job", priority=100, service="PixarRender")
>>> job.newTask(title="A one-command render task", argv=["/usr/bin/prman", "file.rib"], service="pixarRender")
>>> print job.asTcl()
Job -title {a one-task render job} -priority {100.0} -service {PixarRender} -subtasks {
  Task {A one-command render task} -cmds {
    RemoteCmd {{/usr/bin/prman} {file.rib}} -service {pixarRender}
  }
}
```

Spooling a Job

To spool a job, call the job's spool() method. The job id of the new job will be returned.

```
>>> newJid = job.spool()
```

An exception will be raised if there is a problem communicating with the engine. If the job successfully spools, it will appear in the Dashboard or in tq jobs. Otherwise, the engine will log diagnostic information as to why the job could not be spooled.

Spooling can optionally be performed in blocking mode: spool(block=True) will not return until the job submission has been fully processed. If there is an error, an exception will be raised. The block=True option is useful for bringing more diagnostic information to the client. However, it does take longer to complete, and moreso when there are many other jobs concurrently being spooled. As such, blocking is most useful when debugging authoring scripts or developing new ones.

The owner keyword argument can be used to specify the job owner. The default job owner is the script's process owner.

```
>>> job.spool(owner="cher")
```

Setting Attributes

Attributes can be set using keyword arguments (as above) or using attribute assignment. The above job can also be specified as follows:

```
>>> job = author.Job()
>>> job.title = "a one-task render job"
>>> job.priority = 100
>>> job.service = "PixarRender"
>>> job.newTask(title="A one-command render task", argv=["/usr/bin/prman", "file.rib"])
```

Adding Tasks

Adding a task to a job can be done using Job.newTask() as above, or in separate steps of creating a task and adding it as a child of the job.

```
>>> job = author.Job(title="a one-task render job", priority=100, service="PixarRender")
>>> task = author.Task(title="A one-command render task", argv=["/usr/bin/prman", "file.rib"])
>>> job.addChild(task)
```

For tasks to run serially, such that one task finishes before another starts, the task to run first is declared as a child of the second.

```
>>> parent = author.Task(title="parent runs second", argv=["/usr/bin/command"])
>>> child = author.Task(title="child runs first", argv=["/usr/bin/command"])
>>> parent.addChild(child)
```

By default, sibling tasks are permitted to run in parallel.

```
>>> parent = author.Task(title="comp", argv=["/usr/bin/comp", "fg.tif", "bg.tif"])
>>> child1 = author.Task(title="render fg", argv=["/usr/bin/prman", "fg.rib"])
>>> child2 = author.Task(title="render bg", argv=["/usr/bin/prman", "bg.rib"])
>>> parent.addChild(child1)
>>> parent.addChild(child2)
```

Serialsubtasks

Setting a task's serialsubttasks attribute to 1 causes its children to be run serially.

```
>>> parent.serialsubtasks = 1
```

When a parent task has serialsubtasks set, a child task's entire subtree must be fully completed before its sibling task is allowed to run.

Adding Commands

Tasks typically have one or more commands. Adding a command to a task can be done using Task.newCommand(), or done in separate steps of creating a new command and adding it as a child of a task. The following examples build an equivalent task.

```
>>> task = author.Task(title="render rib 1")
>>> command = author.Command(argv=["/usr/bin/prman", "1.rib"], service="pixarRender")
>>> task.addCommand(command)

>>> task = author.Task(title="render rib 1")
>>> command = task.newCommand(argv=["/usr/bin/prman", "1.rib"], service="pixarRender")
```

Additionally, there is a shortcut in which a command can be instantiated and automatically associated with a task when the task is created. This is done by setting the argv attribute when initializing a new task. This approach also works in the author.newTask() method.

```
>>> task = author.Task(title="render rib 1", argv=["/usr/bin/prman", "1.rib"], service="pixarRender")

>>> task = otherTask.newTask(title="render rib 1", argv=["/usr/bin/prman", "1.rib"], service="pixarRender")
```

Because each command requires a service key, the API will push the assignment of the service attribute to the command when this shortcut is used.

If other attributes of a command need to be set, such as the envkey, commands must be built explicitly using author.Command() orauthor.Task. newCommand() instead of using this shortcut.

It is valid for a task to have no commands. For example, a task may serve to group a subtree of tasks, even though the task itself has no commands to execute.

Multi-command Tasks

When a task has multiple commands, they will be executed serially. Multiple commands can be associated with a task using successive invocations of the author.Task.addCommand() or author.Task.newCommand() methods.

```
>>> task = author.Task(title="multi-command task", service="PixarRender")
>>> copyin = author.Command(argv=["scp", "remote:/path/file.rib", "/local/file.rib"])
>>> task.addCommand(copyin)
>>> render = author.Command(argv=["/usr/bin/prman", "/local/file.rib"])
>>> task.addCommand(render)
>>> copyout = author.Command(argv=["scp", "/local/file.tif", "remote:/path/file.tif"])
>>> task.addCommand(copyout)

>>> task = author.Task(title="multi-command task", service="PixarRender")
>>> task.newCommand(argv=["scp", "remote:/path/file.rib", "/local/file.rib"])
>>> task.newCommand(argv=["/usr/bin/prman", "/local/file.rib"])
>>> task.newCommand(argv=["scp", "/local/file.tif", "remote:/path/file.tif"])
```

Exceptions

When assigning a value to an element's attribute, an exception is raised if it is not a valid value.

```
>>> try:
>>>     job.atmost = "three"
>>> except author.AuthorError, err:
>>>     print "you can expect to see this message"
```

Directory Mapping

A directory map is a way to translate file system paths between different operating systems. Directory maps are specified as follows:

```
>>> job.newDirMap(src="X:/", dst="//fileserver/projects", zone="UNC")
>>> job.newDirMap(src="X:/", dst="/fileserver/projects", zone="NFS")
```

More information about directory mapping and this example can be found here.

Engine Connectivity

The author module handles connectivity to the engine when a job is spooled. The engine hostname, port, connected user, password, and debug flag can be changed using setEngineClientParam().

The default target engine is set according to the TRACTOR_ENGINE environment variable; otherwise, tractor-engine:80 is used. It is possible a different default engine is used depending on whether EngineDiscovery has been configured.

The default user is set according to the USER environment variable; otherwise, it is root.

The default password is set according to the TRACTOR_PASSWORD environment variable; otherwise, it is not set.

The default debug flag is set according to the TRACTOR_DEBUG environment variable; othewise, it is False.

The following example demonstrates how to override certain connection values. Note that setEngineClientParam() should be called before spooling any jobs in order for the connection parameters to take effect.

```
>>> author.setEngineClientParam(hostname="test-engine", port=8000, user="donovan", debug=True)
```

You can terminate the session by calling closeEngineClient(). This is helpful for reducing the number of sessions maintained by the engine.

```
>>> author.closeEngineClient()
```

More Help

The file <TractorInstallDir>/lib/python2.7/site-packages/tractor/api/author/test.py contains additional examples of API usage.