Method documentation for `QueueReader-0.9.0`

Methods used by clients

queue

Place a request on the queue.

Syntax:

```
queue request
```

`request` may be anything, but is usually a single line of text.

returns the `id` of the request, in the form of a `unique.timestamp`.

queueFile

Place a request on the queue. The request is either the contents of a file, or `stdin`.

Syntax:

```
queueFile filename
```

or

```
stream | queueFile
```

where *`filename`* is the name of a file.

`stream` is a process that generates the contents of the request.

Returns the `id` of the request, in the form of a `unique.timestamp`.

Note: It is a contents of the file that is placed on the queue, not the filename. Once the request has been queued, the file is no longer required, and the user is free to remove it, or modify it, without having an effect on the queued request.

Methods for processing requests.

You must not use more than one of `doNext` and/or `flushQueue` at any one time. Also, do not use either of them if a `daemon` is running (documented below).

doNext

Process the next (n) request(s) from the queue syntax:

```
doNext [ n ]
```

where *n* is the number of requests to process. *n* defaults to `1`.

flushQueue

Process all requests pending on the queue.

syntax:

```
flushQueue
```

Will continue until no requests are pending.

pause

Pause `flushQueue`. After the current request is completed, `flushQueue` will pause (that is, go to sleep). Use `unpause` to resume processing.

Syntax:

```
pause
```

performRequest

What the QueueReader finally does with the request.

Syntax:

```
performRequest requestFilename
```

The filename is relative to the base directory for the QueueReader.

The default action, implemented by the framework, is to simply stream the contents of the request to `stdout`.

This is one method that is likely to be over-ridden for a clone of **`QueueReader`**. (the other is `daemon`)

processRequest

Process a request from the queue

Syntax:

```
processRequest id
```

The actual sequence is as follows:

1. The request is moved from the `queue/` subdirectory to the `done/` subdirectory. The filename is changed to indicate that processing has commenced.

2. The `performRequest` method is run.

3. The file on the `done/` subdirectory is touched to bring it up to date (using the `Unix` `touch` utility). This is important for archiving, since it provides an indication of how long ago the request was run rather than how long ago the request was queued.

4. The filename on the `done/` subdirectory is modified to show that the request was completed, and to indicate the time that the request was completed.

stop

Stop `flushQueue` after it has finished processing its current request

Syntax:

```
stop
```

unpause

Un-pause `flushQueue`. Will resume `flushQueue` if paused.

Syntax:

```
unpause
```

The daemon

The daemon provides a 'set-and-forget' method for processing methods as they arrive. Once started, the daemon can be stopped, paused and unpaused using the same methods as you would use for `flushQueue`. (See documentation for `stop`, `pause` and `unpause`, above.)

You can pipe the output (`stdout`) from the daemon to another process by setting the `pipeDaemonTo` key and using the `start` method. Once set, the key will be used whenever the `start` method is used.

daemon

Start (in the foreground) a process which will continue to flush the queue on regular basis.

Will not run if a daemon is already running (or appears to be running).

Syntax:

```
daemon
```

daemon.sleep

Used by the daemon to sleep when there are no requests on the queue.

Syntax:

```
daemon.sleep
```

The key `daemon.sleep.time` stores the time (in seconds) that the deamon should sleep before checking the queue again.

kill

Shoot the daemon dead. Be warned, however, there may still be a run-away process left over from `performRequest`.

syntax:

```
kill
```

start

Start the QueueReader daemon as a background process.

Syntax:

```
start
```

Returns the pid of the daemon process.

You can pipe `stdout` from the daemon to another process by using the key `pipeDaemonTo`. Set this key to a shell command. The command will be run, and the daemon output piped to it.

status

Determine the status of the daemon.

Syntax:

```
status
```

Possible responses are:

```
running
running: paused
no daemon running
daemon has probably died. pid was pid
```

wake

Wake the daemon if it is sleeping.

Syntax:

```
wake
```

administration

These methods are useful for someone administering the daemon.

archive

Archive the `done/` queue, by moving any requests older than one day (including one day) to another directory, then tarring that directory.

Syntax:

```
archive [ directory ]
```

`directory` is the name of the directory to be used. It is optionally created. It may be given as an absolute path, or relative to the QueueReader base directory.

If not given, it defaults to `done.`*`YYMMDD`* where *`YYMMDD`* is today's date .

catAll

Show contents of all requests pending

Syntax:

```
catAll
```

catLast

Show the contents of the last (n) request(s) to have been processed

syntax:

```
catLast [ n ]
```

where *n* is the number of requests to show. *n* defaults to `1`.

catNext

Show the contents of the next (n) request(s) from the queue

syntax:

```
catNext [ n ]
```

where *n* is the number of requests to show. *n* defaults to `1`.

clearLockFiles

Clear all the lockfiles, so we can restart the daemon. (Usually required after a power failure, or if the daemon dies unexpectedly.) The lockfiles are the keys created by the method `manageProcess`.

Syntax:

```
clearLockFiles
```

pending

Return the number of requests currently on the queue.

Syntax:

```
pending
```

showAuditTrail

Show the names of last (n) request(s) on the done queue.

Syntax:

```
showAuditTrail [ n ]
```

The output from this method will reveal when each request was originally queued, whether it finished, and the time that it completed.

Requests that did not finish (or are currently being processed) end with `.started`. Requests that completed have `.done.` in their name.

low-level

These methods would not normally be used. They are used internally by the QueueReader.

killManagedProcess

Kill a process that is being "managed" using the `manageProcess` method.

syntax:

```
    killManagedProcess pidKey psKey
```

`pidKey` and *`psKey`* are the names of the keys that were used for the `manageProcess` method.

manageProcess

Manage a running process so we can kill it later if neccessary.

syntax:

```
    manageProcess pid pidKey psKey
```

`pid` is the pid of the running process.

`pidKey` is the key to store the `pid` in.

`psKey` is the key to store the `psQuery` information in.

psQuery

Use `ps` to query a specific pid. (Used to ensures that `start`, `status`, `kill` and `wake` are robust and consistent.)

syntax:

```
    psQuery pid [ verbose ]
```

If the keyword `verbose` is used, then the first line of the `ps` command is included. This line would normally be ignored by scripts that want just the information, so it is removed by default.