Director

Thespian
Director Utility

By: Kevin Quick <quick@sparq.org>

2017 Nov 27 (#1.00)

Thespian Project

TheDoc-07

PUBLIC DOCUMENT

Contents

1 Thespian Director **3**

1.1 Loadable Sources . . . 3

2 Director Functionality **3**

2.1 Invoking the Director Utility . . . 4

2.2 Director Source Authority . . . 4

2.3 Creating Source Packages . . . 4

2.3.1 Long-running Actor Systems . . . 5

2.3.2 Creating a signing key . . . 5

2.3.3 Specifying source inputs . . . 5

2.3.4 Versioning . . . 6

2.3.5 Full gensrc example . . . 7

2.4 Specifying Thespian Configuration Settings . . . 9

2.4.1 Director Log Filter . . . 10

2.4.2 Configuration confirmation . . . 10

2.5 Starting Thespian . . . 10

2.6 Stopping Thespian . . . 11

2.7 Managing loaded sources . . . 11

2.7.1 Loading the latest source versions . . . 12

2.7.2 TLI Files . . . 13

2.8 Director API . . . 14

3 Considerations for using Loadable Sources **16**

3.1 Source Package limitations . . . 16

3.2 Communication across package boundaries . . . 17

3.3 Actor specifications for createActor . . . 17

1 Thespian Director

As described in the `Using Thespian` document, it is possible to use the Thespian Director utility to assist in working with loadable Actor sources. This document describes the Director and associated loading functionality in detail.

It is possible to use the loadable source functionality without using the Thespian Director, but it is much easier when using the common functionality provided by the Director.

1.1 Loadable Sources

As a review, loadable Actor sources provide a method for hot-updates of the Actor source code and possibly running multiple versions of the sources in parallel. This has a number of advantages:

- The sources only need to be loaded into the leading ActorSystem; all other ActorSystems will update from the leading system automatically on an as-needed basis. This also alleviates the need to restart the ActorSystem on the remote nodes each time the source is updated.
- Multiple versions of the sources can be running simultaneously. One use case for this is allowing the older sources to continue running the requests submitted to that source while directing all new requests to the newer version. The older sources can be unloaded at a later time after all of the in-progress requests have been completed.
- Loaded sources are validated by the Source Authority, which can use cryptographic signature validation of the sources, increasing the overall security of the system.

2 Director Functionality

The Thespian Director is made up of several components:

1. A command-line interface allowing configuration and management of the loadable sources.
2. A source package creator that generates a signed, loadable source package from a set of inputs.
3. A Director Actor which maintains a list of the currently loaded sources and assists in loading new sources.
4. A Source Authority Actor which validates loaded sources.
5. Configuration specifications for starting up a Thespian ActorSystem with various configuration settings. These specifications are designed to be easily updated via automated means to support simple configuration updates across an entire environment.
    - This includes configuration of the logging subsystem for Thespian interaction.
6. Common logging filter function to ensure logging can display actor addresses consistently.
7. Thespian startup using the specified configuration settings. This startup can be performed on demand or it can be configured to be started at system boot time for systemd, initctl, and Windows-based operating systems.

2.1 Invoking the Director Utility

The Thespian Director is easily invoked from the commandpline via the python module specification syntax:

```
$ python -m thespian.director [verbose] [command [args...]]
```

When invoked with no command, the director will output help information to stdout but take no other action. This can be used to quickly determine what the needed command is or the overall settings are.

The Director takes several primary commands, and each has a different set of arguments. The "help" command will print the same information as when specifying no command, but following the help command with the name of another command will print help information specific to that second command.

When invoked, the Director will typically look for various local files specifying configuration settings and loadable sources. It is customary to setup a single directory to contain all of these files, and the Director uses the THESPIAN_DIRECTOR_DIR environment variable to locate this directory; if not set, the directory defaults to "/opt/thespian/director" under Unix or "C:thespian" under Windows.

2.2 Director Source Authority

One of the primary purposes of the Director is to support the ability to dynamically load new Actor sources into a running Thespian ActorSystem (or a Convention of ActorSystems). In order to safely load these sources and provide appropriate security, Thespian will not use any loaded source unless the registered Source Authority indicates that the source has been validated.

The Director provides a Source Authority actor that will register as the validator of these packages. The Source Authority will ensure that the loaded source package was generated by the Director itself (see the next section) and that the source itself is signed by a cryptographic key that validates the provenance of the source and that the source has not been modified after it was signed. To do this, the Source Authority will read any public key files (named "*_tls.key") found in the THESPIAN_DIRECTOR_DIR location and use these to verify the loaded sources. There may be multiple key files and these key files are read dynamically each time a source is loaded; as long as a source can be validated by one of the key files, it will be accepted.

The Director's Source Authority is started automatically when the Thespian ActorSystem is started by the Director. Other than ensuring the correct key files are available in the Director's operational directory there is no configuration or ongoing management needed for the Source Authority.

2.3 Creating Source Packages

In order to create a loadable source, the varous Python files must be packaged into a ZIP file, and that ZIP file should be prepared for validation by the Source Authority. This preparation process will modify the ZIP file, making it invalid in such a way that it can only be restored and used if the Source Authority successfully validates the loaded file.

The preparation process will then sign the sources with a locally-specified private key file. The public key corresponding to this private key must be placed in the Directors operational directory on each system where the loaded

sources may be utilized (each ActorSystem in a convention will independently validate the loaded sources when utilized). The private key should **not** be distributed in this manner: any entity possessing the private key can create a loadable source that will be accepted by the Source Authority. The private key should only be accessible to the system where the loadable sources are built or generated.

2.3.1 Long-running Actor Systems

The overall operational concept of using the Director and associated loadable sources is that the Thespian ActorSystem is started up at boot time or the first time it is needed, but that it is thereafter left active on the system. The individual sets of Actors that are needed are loaded as a source package when that need arises. Loaded sources may be unloaded when no longer needed, and newer versions can be loaded, all without shutting down the ActorSystem itself.

Multiple different Actor source packages can be loaded at any one time. Those packages can be newer versions of existing packages, or they can be completely independent packages. This allows multiple different and independent Actor-based applications to be run without requiring consolidation or combination of those applications. Each application can even be independently generated using a separate private key: as long as one of the public keys available to the running Director Source Authority matches the loaded source that loadable source will be useable.

Each source package will be identified by a "group" name; a newer package that should replace an older package will have the same group name, but independent packages will have different group names.

Older loaded sources can be unloaded on-demand or automatically based on configuration specifications discussed in later parts of this document.

2.3.2 Creating a signing key

The customary way to create a key pair for signing sources is to use the openssl toolset. As an example:

```
$ openssl genrsa -out sourcekey.pem 1024
$ openssl rsa -in sourcekey.pem -pubout -out sourcekey_tls.key
```

This would create a 1024-bit private key file `sourcekey.pem`, and an associated public key file `sourcekey_tls.key`. The public key file would be installed in the THESPIAN_DIRECTOR_DIR location on all target machines, and the private key file is used to sign the generated loadable sources.

There are many parameters that can be used when creating a keypair; please consult the openssl documentation for more information.

2.3.3 Specifying source inputs

Once a keypair has been created (which is usually a one-time operation), signed loadable source packages can be generated using the "gensrc" Thespian Director command:

```
$ python -m thespian.director gensrc ...
```

The specific set of arguments are summarized by the help information output for this command:

```
$ python -m thespian.director help gensrc
```

One parameter is the `sources_dir`, which specifies the root path to the sources. All loaded sources are expected to be found underneath this root directory path, and this directory path will **not** be part of the file tree existing within the loadable ZIP file.

One or more source files must be specified to be included in the ZIP file. Each of these source specifications can use the Python `glob.glob` specification syntax, but remember to use quotes to avoid conflicts with shell wildcard expansion (note: Thespian versions after 3.8.3 running under Python 3.5 or later can use the "**" glob specification to recurse into subdirectores). Alternatively, each file can be explicitly specified and shell wildcarding can apply.

In addition to local files, dependent packages can be included in the ZIP file as well. To specify the list of packages that should be included, create a file that lists the names of these packages, one per line. Unlike a pip requirements file, there should be no specification of package versions: the gensrc does not download packages but instead collects the named packages as previously installed into the local environment, so this file should be viewed as a complement to whatever tool/specification is used to install those packages. Having a separate package list also allows a distinction between packages needed for the running Actor application and which should be included in the loadable sources as separate from any packages installed to facilitate local debugging (e.g. pyflake) which are not needed at runtime by the Actors. This dependency file is provided to gensrc as an additional argument, identified by the "deps:" prefix.

Each loadable source group also uses an information file that specifies the runtime behavior and configuration of the loadable source group should be handled by the Director. This information file is called a "TLI" file and it is described in more detail along with the corresponding functionality in the sections below. The TLI file is specified to the "gensrc" command as an additional argument, identified by the "tli:" prefix.

2.3.4 Versioning

One of the arguments to the gensrc command specifies the version of the loadable source file. As described earlier, updated sources can be loaded to replace older versions of those sources. To identify which loadable sources are newer than others within a group, the "version" of the loadable source file is used.

The Thespian Director allows a version to be specified on the gensrc command line as an additional argument identified by the "version:" prefix. Thespian will use a numerical sorting algorithm for each field, where fields are separated by periods to determine which versions are "greater" or "lesser" than other versions. The "avail" command (described in later sections) can be used to see what order the Director perceives the existing versioned files to be in.

The word "date" can be used as a special version; the actual version applied to the loadable source file generated will have an ISO date format of "YYYYMMDDHHMM". This form is advantageous in that the resulting versions will sort in the same order as the gensrc commands were chronologically run, which is usually the proper sense of code progression. Although it is perfectly reasonable to distribute loadable sources using date-based versions, one common usage pattern is to use dates during development processes and then use the actual software release

version (based on whatever local versioning scheme is used) at release time. The date-based versions create very large numbers which are almost always larger than any normal product versioning scheme, ensuring that local development builds are preferred over production releases without requiring frequent production version updates during that development.

It is also possible to generate loadable sources without assigning a version, but this precludes the ability to load multiple versions of the same source at the same time.

2.3.5 Full gensrc example

The following is a hypothetical project tree for an Actor-based application:

```
/home/joedev/projects/foo/setup.py
                         /__init__.py
                         /fooactor.py
                         /subdir1/__init__.py
                         /subdir1/baractor.py
                         /subdir2/__init__.py
                         /subdir2/barn/__init__.py
                         /subdir2/barn/cowactor.py
                         /subdir2/barn/pigactor.py
                         /requirements.txt
                         /deppkgs.txt
                         /foo.tli
```

The `requirements.txt` file is a typical specification that allows the `pip` tool to install the local dependent packages:

```
$ cat requirements.txt
tabulate==0.75
jsonschema==2.5.1
Jinja2==2.9.*
pytest
pytest-benchmark >= 3.0
$
```

The `deppkgs.txt` file is used to specify the package dependencies for the Thespian loadable source (which does not include the test packages):

```
$ cat deppkgs.txt
tabulate
jsonschema
Jinja2
$
```

A loadable source package using a date-based version can be created with the following command:

```
$ cd /home/joedev/projects

$ ls /opt/thespian/director
sourcekey_tls.key

$ python -m thespian.director gensrc foo /home/joedev/sourcekey.pem \
         /home/joedev/projects version:date \
         deps:./deppkgs.txt \
         foo/*.py $(find foo/subdir1 -name '*.py') 'foo/subdir2/**/*.py'

$ ls /opt/thespian/director
foo-201710261902.tls foo.tli sourcekey_tls.key

$
```

The tls file that was generated will contain the python files from `/home/joedev/projects/foo`, as well as the `tabulate`, `jsonschema`, and `Jinja2` packages. The tls file is therefore a fully self-contained actor application.

A .tls file can be verified by using the `tlsinfo` Director command:

```
$ python -m thespian.director tlsinfo /opt/thespian/director/foo-201710261902.tls
Validated "/opt/thespian/director/foo-201710261902.tls" loadable with key /opt/theshpian/director/sou

$
```

This ensures that the provided public key can be used by the Director's Source Authority to successfully validate the .tls loadable source file.

The table-of-contents of a .tls file can be obtained as well by adding the argument "contents", and the source of a particular loaded file can be obtained by specifying that filename as an argument.

```
$ python -m thespian.director tlsinfo /opt/thespian/director/foo-201710261902.tls contents
   4317 foo/fooactor.py
    111 foo/__init__.py
  62315 foo/subdir1/baractor.py
   ...

$ python -m thespian.director tlsinfo /opt/thespian/director/foo-201710261902.tls foo/fooactor.py
import sys
import os
import jsonschema

main_schema = 'blah'
...

$
```

These commands can help ensure that the loadable source package contains the expected contents.

2.4 Specifying Thespian Configuration Settings

One of the Director's capabilities is being able to startup the Thespian ActorSystem on the local host to be able to run the Actor applications. In order to properly start the ActorSystem, the Director needs the set of configuration parameters that should be used for that startup.

The configuration elements used for the ActorSystem are read from files in the $THESPIAN_DIRECTOR_DIR directory. These configuration elements are **not** in a "config.ini" style file, nor are they JSON, YAML, or any other encoding. Instead, each configuration value is maintained in a separate file whose value can simply be read and directly used. This is done to make it easy to update these configuration files via automated means: no existing configuraiton format must be read and partially manipulated: simply re-write an entire file with the new configuration value.

The configuration files and their usage are described by running the `help` command for the Director:

```
$ python -m thespian.director help
```

Here is an example that specifies a remote convention leader, a local set of capabilities, a logging configuration, and a logging directory.

```
$ ls $THESPIAN_DIRECTOR_DIR
convleader.cfg othercaps.cfg thesplog.cfg thesplogd.cfg

$ cat convleader.cfg
192.168.32.19

$ cat othercaps.cfg
{ 'Worker': True, 'Slot': 19, 'nproc': 2 }

$ cat thesplog.cfg
{ 'version': 1,
  'formatters': {'normal':
     {'fmt': '%(asctime)s,%(msecs)d %(actorAddress}s/PID:%(process)d %(name)s %(levelname)s %(message)
      'datefmt': '%Y-%m-%d %H:%M:%S'}},
  'root': {'level': 20, 'handlers': ['foo_log_handler']},
  'loggers': {'subsysA': {'level':30, 'propagate': 0, 'handlers': ['foo_log_handler']}},
  'handlers': {
    'foo_log_handler': {
       'filename': 'fooapp.log',
       'class': 'logging.handlers.TimedRotatingFileHandler',
       'formatter': 'normal',
       'level': 20, 'backupCount':3,
       'filters': ['isActorLog'],
       'when': 'midnight'}}
```

```
  'filters': {'isActorLog': {'()', 'thespian.director.ActorAddressLogFilter'}}
}

$ cat thesplogd.cfg
/tmp/log

$
```

2.4.1 Director Log Filter

As seen in the example configuration in the previous section, the `thespian.director.ActorAddressLogFilter` can be specified as the filter for logging. This filter is a convenience filter supplied by the Director which ensures that the "`actorAddress`" formatting parameter is defined for messages. This parameter will contain the address of the Actor which generated the message, or an empty string if the message did not come from an Actor.

2.4.2 Configuration confirmation

The `config` Director command can be used to echo various information about the configuration observed by the Director:

```
$ python -m thespian.director config
   Sources location: /opt/thespian/director/
        System Base: multiprocTCPBase
         Admin Port: 1900
  Logging directory: /tmp/log
 Other Capabilities: { "Worker": True, "Slot": 19, "nproc": 2 }

$
```

This command can be used to verify where the Director is looking for configuration information and that it is able to parse the basic configuration specifications correctly.

2.5 Starting Thespian

The Director can be used to start the local Thespian ActorSytem either on-demand or at boot time. The startup will use the configuration parameters defined in the configuration files described above, and the startup will have no effect if an ActorSystem is already running.

To manually start:

```
$ python -m thespian.director start
```

To configure Thespian to be started at boot time:

```
$ python -m thespian.director bootstart
```

Note that the latter command will also start Thespian immediately as well unless the word "nostart" appears on the line after the bootstart command. There may also be a name argument after the bootstart (and before the "nostart" argument) which will specify the name to assign to the system service associated with the Thespian ActorSystem started at boot time; the default is "thespian.director".

The boot configuration automatically detects the current system's startup manager and writes the appropriate files to the appropriate locations. Currently supported system managers are: systemd, initctl, and Windows Boot Manager.

2.6 Stopping Thespian

A running Thespian Actor System can be stopped with the `shutdown` Director command:

```
$ python -m thespian.director shutdown
```

This will cause all Actors to receive an ActorExitRequest and the Actor System itself to shutdown, just as if the `ActorSystem().shutdown()` method had been called.

2.7 Managing loaded sources

The source files generated by the 'gensrc' command (described above) can be loaded into the current Thespian ActorSystem and run. The Director can be used to load individual sources on-demand, or load the latest version of each source in every group.

To load a single source, specify the name of either the tli file or the specific tls file to load:

```
$ ls /opt/thespian/director
foo-201710261938.tls foo-201710261902.tls foo.tli    sourcekey_tls.key
other-1.3.5          other-1.4            other-2.0  other.tli

$ python -m thespian.director load foo-201710261938.tls

$ python -m thespian.director load foo
```

Both of the load commands above are equivalent and will load the indicated tls source into the Thespian ActorSystem. The load command will automatically start the Thespian ActorSystem if it is not currently running.

The set of currently loaded and running sources can be retrieved with the `list` command:

```
$ python -m thespian.director list
```

Each loaded source is described by its group name and also by its hash value. The hash value is the same hash value returned by the `ActorSytstem().loadSource()` function and represents the hash of the loaded source.

The sources can be unloaded as well:

```
$ python -m thespian.director unload {HASH}
```

The source associated with the specified hash value is unloaded.

To find out what sources are available to be loaded, use the `avail` Director command:

```
$ python -m thespian.director avail
THESPIAN_DIRECTOR_DIR=/opt/thespian/director
foo: /opt/thespian/director/foo-20170261938.tls  4d41d823c3702e9cf7b8b5a650429d7e
   + /opt/thespian/director/foo-20170261902.tls  2a9fcdc7589812b4596f317e85b76042
other: /opt/thespian/director/other-2.0.tls    1bd9df7222c035855babefdfff8f9e29
     + /opt/thespian/director/other-1.4.tls    8406cf8f4596f9cfcae33c7f463bfc1f
     + /opt/thespian/director/other-1.3.5.tls  d9550b9af01692d6a4154cc54cd8d9aa

$
```

The above shows sample output for the directory contents previously shown. It lists each loadable source by group, sorted by version within the group with the most recent version first and the oldest version last. Each line also specifies the hash associated with the loadable source.

To remove a loadable source from consideration for loading, simply delete it from the `THESPIAN_DIRECTOR_DIR` directory.

2.7.1 Loading the latest source versions

The Director `load` command is useful for loading a specific source, and also automatically selecting the latest source if just the group name is specified, but for many uses it is normal to load the latest source load for **all** groups.

To automatically load the latest source for every group, use the Director `refresh` command:

```
$ python -m thespian.director refresh
Loaded "foo" /opt/thespian/director/foo-2017021938.tls: 4d41d823c3702e9cf7b8b5a650429d7e
Loaded "other" /opt/thespian/director/other-2.0.tls    1bd9df7222c035855babefdfff8f9e29

$
```

As with the `load` command, the `avail` command will also startup a Thespian Actor System if one is not already running. When the `bootstart` command has been used to automatically startup Thespian at boot time, the boot script will issue the `refresh` command to start Thespian and load all of the latest sources.

If the latest source for a group is already loaded, the `refresh` has no effect on that group.

2.7.2 TLI Files

When loading a loadable source, it is also typical to perform various startup operations for the source that has been loaded, and it is also convenient to perform shutdown operations for the previously active source (loading a new version of a source does not unload the previous version—unless it has been configured to do so—but the previous running instance may still wish to perform shutdown activities).

The TLI file that is associated with each group contains the specification of these startup and shutdown activities.

Here is an example of a TLI specification:

```
$ cat $THESPIAN_DIRECTOR_DIR/foo.tli
{ 'Actors':
    { 'foo.fooactor':
        { 'OnLoad': { 'Role': 'MainFoo',
                      'Message': 'StartFoo',
                      'GlobalName': 'Foo Primary',
                    },
          'OnReactivate': { 'Message': ('Restart', True) },
          'OnDeactivate': { 'Message': {'Shutdown': True,
                                        'timeout': 3 } },
        },
      'foo.subdir2.barn.cowactor':
        { 'OnLoad': { 'Role': 'FlyingCow',
                      'Message': 'Moo!'
                    },
        },
    },
}

$ cat $THESPIAN_DIRECTOR_DIR/other.tli
{
  'Actors':
    { 'other.mainActor': { 'OnLoad': { 'Role': 'other',
                                       'Message': 0 }}
    }
  'AutoUnload': True,
  'TLS_Keep_Limit': 3,
}

$
```

The contents of each TLI file is a Python dictionary, where the primary key is "Actors". The value for that key is another dictionary, whose keys are the import path specifications of Actors within the loaded source.

The value for each of those dictionaries can specify 'OnLoad', 'OnReactivate', and 'OnDeactivate' keys. When a new loadable source is loaded, each actor specified is created by the Director, using the GlobalName (if specified), and sent the specified message (if any). The "Role" specifies an associated name for that Actor that can be retrieved

with other Director commands. Both the 'Message' and the 'GlobalName' key are optional.

If a previous source was loaded for this group when the new source was loaded, the previous source is sent the 'Message' specified in the 'OnDeactivate' section (if there is one). The only recognized element of the 'OnDeactivate' section is the 'Message' key. As stated above, the old source is not necessarily unloaded unless the 'AutoUnload' primary key is present and its corresponding value is True; the 'OnDeactivate' message is always sent, but the subsequent unload only occurs in the AutoUnload case.

No imports can be performed for this file, so the Message sent can only be a builtin Python type, or a value from the `sys` or `os` modules.

If an old loaded source was superceded by a new loaded source, and then the new loaded source is subsequently unloaded, the highest loaded version of the older, still loaded source is re-activated. When this occurs, the 'OnReactivate' message is sent to the older source's actor.

When the `unload` Director command is specified, the 'OnDeactivate' operation will be performed and then the specified Actors will be sent an `ActorExitRequest()` message.

If the `shutdown` director command is used, no 'OnDeactivate' message will be sent and (all) the loaded actors will simply receive the `ActorExitRequest()` message.

The `TLS_Keep_Limit` key is also optional, but affects the contents of the `THESPIAN_DIRECTOR_DIR` instead of the running Thespian environment: when a load (or refresh-caused load) has completed, the Director will determine the number of .tls sources in the `THESPIAN_DIRECTOR_DIR` for that group, deleting older files to reduce the number of loadable sources for that group to the `TLS_Keep_Limit`. This allows old loadable source distributions to be automatically cleaned up and removed.

2.8 Director API

As well as the command-line interface, a Python program can interact with the Director in two ways:

1. Exchanging messages to the Director Actor (via `ActorSystem.ask()` or another Actor).
2. Via the DirectorControl object.

In the first form, the source for the director can be examined: there are documentation strings describing all the messages that can be exchanged with the Director and the associated responses. The Director Actor is registered under the "Director" global name, so an example of retrieving information about all loaded sources is as follows:

```
import thespian.actors
asys = ActorSystem()
director_addr = asys.createActor('thespian.director', globalName='Director')
resp = asys.ask(director_addr, { "DirectorOp": "RetrieveAll"})
print(str(resp))
```

The second alternative is to create an instance of the DirectorControl object. Once created, there is a `cmd_COMMAND` method for each COMMAND that the Director supports on the command-line, along with arguments corresponding to the arguments supplied to the command-line version. There is also docstring help for each method.

```
$ THESPIAN_DIRECTOR_DIR=/opt/thespian/director python3
>>> from thespian.director import DirectorControl
>>> dc = DirectorControl()
>>> dc.cmd_avail()
THESPIAN_DIRECTOR_DIR=/opt/thespian/director
foo: /opt/thespian/director/foo-20170261938.tls  4d41d823c3702e9cf7b8b5a650429d7e
   + /opt/thespian/director/foo-20170261902.tls  2a9fcdc7589812b4596f317e85b76042
other: /opt/thespian/director/other-2.0.tls    1bd9df7222c035855babefdfff8f9e29
     + /opt/thespian/director/other-1.4.tls    8406cf8f4596f9cfcae33c7f463bfc1f
     + /opt/thespian/director/other-1.3.5.tls  d9550b9af01692d6a4154cc54cd8d9aa
0
>>> help(dc.cmd_avail)
... docstring information ...
```

The final `0` in the above output is the return value from the command method: command methods will return 0 on success or non-zero on error.

There is an additional method: the `DirectorControl.get_role_address('ROLENAME')` method, which does not print any output but returns a dictionary with information about the currently active Actor associated with the specified Role (as specified in the TLI file).

```
$ THESPIAN_DIRECTOR_DIR=/opt/thespian/director python3
>>> from thespian.director import DirectorControl
>>> dc = DirectorControl()
>>> r = dc.get_role_address('MainFoo')
>>> print(r)
{'DirectorResponse': 'RoleAddress',
 'ActorAddress': <thespian.actors.ActorAddress object at 0x7f266ct1d1d0>,
 'Group': 'foo',
 'ActorClass': 'foo.fooactor',
 'SourceHash': '4d41d823c3702e9cf7b8b5a650429d7e',
 'Role': 'MainFoo',
 'Success': True}
>>> r = dc.get_role_address('unknown role name')
RoleAddress ERROR: {'DirectorResponse':'RoleAddress',
                    'ActorAddress': None, 'Role': 'un',
                    'SourceHash': None, 'Group': None,
                    'ActorClass': None, 'Success':False}
>>> print(r)
None
>>> r = dc.get_role_address('unknown role name', silent=True)
>>> print(r)
None
>>>
```

This method can be used to easily obtain these role-identified actor addresses to initiate communications with them from an application. Printing of errors can be suppressed by passing the argument `silent=True`, and a `group`

argument can also be specified to limit the role search to the corresponding group (otherwise all groups are searched for a role match).

3 Considerations for using Loadable Sources

Loadable sources are a great way to dynamically manage and update one or more Thespian Actor-based applications, and especially for installations spanning multiple systems. There are some limitations for using this loadable source methodology however.

This restriction also applies across loaded source boundaries, but within a loaded source, target actors can be specified by either string or class reference.

3.1 Source Package limitations

Although the `loadActorSource()` functionality works quite well, there are a few scenarios that can cause difficulties:

1. Loaded sources containing or relying on object libraries. The `loadActorSource()` functionality affects the Python module lookup, but it does not affect object file lookups that would need to be performed for native object libraries. In addition, object libraries are specific to their build environment and may not work in all target environments. If the loaded source simply **refers** to an externally available library (e.g. PyOpenSSL python package refers to `/usr/lib/libssl.so`) then this may work if the proper object library has been installed on the target system by other means (i.e. not via the `loadActorSource()` call.

2. Sources containing other PEP302 path importers may not work correctly when used in conjuction with the `loadActorSource()` PEP302 importer. As an example, the `loadActorSource()` importer prefixes loaded modules with the Source Hash associated with those modules to ensure uniqueness. This is not always compatible with other PEP302 importers.

   (a) The `six` package (providing Python2/Python3 compatibility) uses a PEP302 importer to manage moved packages. This functionality of the `six` package is broken by the `loadActorSource()` Source Hash prefixing. To avoid this `loadActorSource()` processing handles `six` as a special case and *does not* prefix `six` with any Source Hash.

   The assumption is that `six` is a mature and unchanging package and a global version is acceptable for all loaded sources (and that a working `six` importer is preferrable to a non-working importer).

3. The `importlib` package has a subtle flaw that is not compatible with the Source Hash prefixing performed by `loadActorSource()`. To work around this problem Thespian provides an updated version of `importlib`; use " `import thespian.importlib as importlib` " instead of just " `import importlib` " in Actor sources that will be loaded via `loadActorSource()`.

4. Loaded sources should expect that `__file__` or other internals can be set to None.

5. It is not possible to send a pickled message between sources, even if that message is imported from the same file/module in both loaded sources. This is because each loaded source (and all of its top level symbols) are internally prefixed with the hash of the source they are contained in; this helps preserve independence

between loaded source modules and prevents interference between them, but it means that the pickled object will have a different source hash on the target side than on the sender side. The object should be encoded in a more conventional format (e.g. JSON).

3.2 Communication across package boundaries

By default, Thespian uses the pickle module to send messages from one actor to another. This requires that the unpickling operation in the target Actor's environment be able to reconsitute the pickled object by name.

When using loadable sources, all names are prefixed with the hash value of the source they have come from, thereby creating a private namespace for each loaded source. Actors within the same loaded source will share the same hash and can use internally created class objects to communicate, but that they will not be able to do this when communicating with Actors in other loaded sources (or the external environment). This means that all messages exchanged with Actors across a loaded source boundary must be regular, built-in Python objects (strings, numbers, booleans, tuples, lists, dictionaries, etc.).

If there is an attempt to send an internal source-loaded object across this boundary, a deserialization error will be written to the logfile for the destination actor, but it will not even be able to generate a PoisonMessage response because it cannot parse the message to determine the sender.

3.3 Actor specifications for createActor

When using the `ActorSystem().createActor()` method, the Actor to be created must be specified by the fully module-qualified string name of the Actor instead of the class reference.

The following two will **not** work:

```
import foo
fail1 = ActorSystem().createActor(foo.fooActor.FooActor)

from foo.fooActor import FooActor
fail2 = ActorSytem().createActor(FooActor)
```

however, the following will work correctly:

```
good_addr = ActorSytem().createActor('foo.fooActor.FooActor', ...)
```