txampext Documentation

Release 20121226

the txampext authors

November 22, 2013

Contents

Contents:

Exposed box senders and protocols

Normally, AMP responders operate in a vacuum. They take some arguments and return a properly formatted response. They don't know where the arguments are coming from, or where the response is going to.

Normally, that's a very useful abstraction. Sometimes, you would like command locators to have access to the box sender or the protocol. For example:

- Your responder may want to switch the box receiver.
- Your responder may want to know information about the transport.

In vanilla AMP as implemented by Twisted, the box sender and the protocol are always the same object, an instance of the `BinaryBoxProtocol` class. The distinction made here has two advantages:

- It also works for more exotic AMP setups
- It conveys intent about what you want to do

1.1 Exposed box senders

In your command definition, add `txampext.exposed.EXPOSED_PROTOCOL` to the arguments. Your responder will be called with a `boxSender` argument, which is the box sender.

Exposing the current box sender requires cooperation from the protocol.

1.1.1 Example

```
from twisted.protocols import amp
from txampext import exposed

class CommandWithExposedBoxSender(amp.Command):
    arguments = [exposed.EXPOSED_BOX_SENDER]
    response = []



class Locator(amp.ResponderLocator):
    @CommandWithExposedBoxSender.responder
    def responder(self, boxSender):
        # Do something with the box sender here...
        return {}
```

TODO: demo how to get access to the box sender

1.2 Exposed protocols

In your command definition, add `txampext.exposed.EXPOSED_PROTOCOL` to the arguments. Your responder will be called with a `protocol` argument, which is the current protocol instance.

Exposing the protocol requires no cooperation from the protocol, and works with vanilla AMP classes.

1.2.1 Example

```
from twisted.protocols import amp
from txampext import exposed

class CommandWithExposedProtocol(amp.Command):
    arguments = [exposed.EXPOSED_PROTOCOL]
    response = []



class Locator(amp.ResponderLocator):
    @CommandWithExposedProtocol.responder
    def responder(self, protocol):
        # Do something with the protocol here...
        return {}
```

Argument constraints

AMP provides many basic argument types, such as byte strings, text strings, integers, et cetera. Sometimes, you want additional constraints, for example, integers between 0 and 100.

2.1 Using constraints

The `ConstrainedArgument` class takes a base argument and zero or more constraints. The base argument is passed as the first positional argument to `ConstrainedArgument`, the constraints are passed as subsequent arguments. It can be used as a regular AMP argument in a command definition.

For example:

```
ConstrainedArgument(Integer, InSet(set([1, 2, 3, 4, 5])))
```

2.2 Built-in constraints

A few simple constraints are shipped with this module.

2.3 Creating your own constraints

A constraint is a very simple callable that takes the actual argument value and returns `True` if the constraint is satisfied, and `False` otherwise. This makes it very easy to write your own constraints, for example:

```
def divisibleBy11(value):
    return value % 11 == 0
```

CHAPTER 3

Multiplexed streams

You can multiplex virtual stream transports over AMP. This means that there are multiple virtual stream connections, where the actual bytes are being transported over an existing AMP connection.

Unlike TCP, these virtual stream transports are symmetric, so there is no “half-close”.

Multiplexed connections use the following simple AMP commands:

Obviously, the AMP peer playing the role of the server in a stream connection must implement all three. The obvious way to do that is to use the `MultiplexingCommandLocator` command locator.

The AMP peer playing the role of a client does not have to implement the `Connect` command, of course. However, since the server side can disconnect, the client must implement `Disconnect`. Since the stream transport is bidirectional, the client must of course also implement a `Transmit` responder, to receive data coming from the server.

3.1 Local proxying

TODO: document (see docs/examples/)

CHAPTER 4

Indices and tables

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

Python Module Index