

---

# **SleekXMPP Documentation**

***Release 1.0***

**Nathan Fritz, Lance Stout**

September 28, 2013



# CONTENTS

<b>1</b>	<b>Here's your first SleekXMPP Bot:</b>	<b>3</b>
<b>2</b>	<b>Getting Started (with Examples)</b>	<b>5</b>
2.1	SleekXMPP Quickstart - Echo Bot . . . . .	5
2.2	Sign in, Send a Message, and Disconnect . . . . .	13
2.3	Create and Run a Server Component . . . . .	17
2.4	Manage Presence Subscriptions . . . . .	20
2.5	Mult-User Chat (MUC) Bot . . . . .	20
2.6	Enable HTTP Proxy Support . . . . .	26
2.7	Send a Message Every 5 Minutes . . . . .	29
2.8	Send/Receive IQ Stanzas . . . . .	29
<b>3</b>	<b>Tutorials, FAQs, and How To Guides</b>	<b>33</b>
3.1	Supported XEPS . . . . .	33
3.2	Following <i>XMPP: The Definitive Guide</i> . . . . .	33
3.3	How to Work with Stanza Objects . . . . .	37
3.4	Creating a SleekXMPP Plugin . . . . .	37
3.5	How to Use Stream Features . . . . .	48
3.6	How SASL Authentication Works . . . . .	48
3.7	Using Stream Handlers and Matchers . . . . .	48
3.8	Plugin Guides . . . . .	48
<b>4</b>	<b>SleekXMPP Architecture and Design</b>	<b>53</b>
4.1	SleekXMPP Architecture . . . . .	53
4.2	Plugin Architecture . . . . .	56
<b>5</b>	<b>API Reference</b>	<b>57</b>
5.1	Event Index . . . . .	57
5.2	ClientXMPP . . . . .	62
5.3	ComponentXMPP . . . . .	65
5.4	BaseXMPP . . . . .	66
5.5	Exceptions . . . . .	74
5.6	Jabber IDs (JID) . . . . .	75
5.7	Stanza Objects . . . . .	76
5.8	Stanza Handlers . . . . .	91
5.9	Stanza Matchers . . . . .	93
5.10	XML Stream . . . . .	94
5.11	Scheduler . . . . .	104
5.12	XML Serialization . . . . .	106

5.13	Python 2.6 File Socket Shims . . . . .	107
5.14	Core Stanzas . . . . .	107
5.15	Plugins . . . . .	107
<b>6</b>	<b>Additional Info</b>	<b>109</b>
6.1	Glossary . . . . .	109
6.2	License (MIT) . . . . .	109
<b>7</b>	<b>Credits</b>	<b>113</b>
	<b>Python Module Index</b>	<b>115</b>

**Get the Code**

```
pip install sleekxmpp
```

The latest source code for SleekXMPP may be found on [Github](#). Releases can be found in the `master` branch, while the latest development version is in the `develop` branch.

**Latest Stable Release**

- [1.0](#)

**Develop Releases**

- [Latest Develop Version](#)

A mailing list and XMPP chat room are available for discussing and getting help with SleekXMPP.

**Mailing List** [SleekXMPP Discussion on Google Groups](#)

**Chat** [sleek@conference.jabber.org](mailto:sleek@conference.jabber.org)

SleekXMPP is an *MIT licensed* XMPP library for Python 2.6/3.1+, and is featured in examples in [XMPP: The Definitive Guide](#) by Kevin Smith, Remko Tronçon, and Peter Saint-Andre. If you’ve arrived here from reading the Definitive Guide, please see the notes on updating the examples to the latest version of SleekXMPP.

SleekXMPP’s design goals and philosophy are:

**Low number of dependencies** Installing and using SleekXMPP should be as simple as possible, without having to deal with long dependency chains.

As part of reducing the number of dependencies, some third party modules are included with SleekXMPP in the `thirdparty` directory. Imports from this module first try to import an existing installed version before loading the packaged version, when possible.

**Every XEP as a plugin** Following Python’s “batteries included” approach, the goal is to provide support for all currently active XEPs (final and draft). Since adding XEP support is done through easy to create plugins, the hope is to also provide a solid base for implementing and creating experimental XEPs.

**Rewarding to work with** As much as possible, SleekXMPP should allow things to “just work” using sensible defaults and appropriate abstractions. XML can be ugly to work with, but it doesn’t have to be that way.



# HERE'S YOUR FIRST SLEEKXMPP BOT:

```
import logging

from sleekxmpp import ClientXMPP
from sleekxmpp.exceptions import IqError, IqTimeout

class EchoBot(ClientXMPP):

    def __init__(self, jid, password):
        ClientXMPP.__init__(self, jid, password)

        self.add_event_handler("session_start", self.session_start)
        self.add_event_handler("message", self.message)

        # If you wanted more functionality, here's how to register plugins:
        # self.register_plugin('xep_0030') # Service Discovery
        # self.register_plugin('xep_0199') # XMPP Ping

        # Here's how to access plugins once you've registered them:
        # self['xep_0030'].add_feature('echo_demo')

        # If you are working with an OpenFire server, you will
        # need to use a different SSL version:
        # import ssl
        # self.ssl_version = ssl.PROTOCOL_SSLv3

    def session_start(self, event):
        self.send_presence()
        self.get_roster()

        # Most get_*/set_* methods from plugins use Iq stanzas, which
        # can generate IqError and IqTimeout exceptions
        #
        # try:
        #     self.get_roster()
        # except IqError as err:
        #     logging.error('There was an error getting the roster')
        #     logging.error(err.iq['error']['condition'])
        #     self.disconnect()
        # except IqTimeout:
        #     logging.error('Server is taking too long to respond')
```

```
        #         self.disconnect()

def message(self, msg):
    if msg['type'] in ('chat', 'normal'):
        msg.reply("Thanks for sending\n%(body)s" % msg).send()

if __name__ == '__main__':
    # Ideally use optparse or argparse to get JID,
    # password, and log level.

    logging.basicConfig(level=logging.DEBUG,
                        format='%(levelname)-8s %(message)s')

    xmpp = EchoBot('somejid@example.com', 'use_getpass')
    xmpp.connect()
    xmpp.process(block=True)
```



---

# GETTING STARTED (WITH EXAMPLES)

## 2.1 SleekXMPP Quickstart - Echo Bot

---

**Note:** If you have any issues working through this quickstart guide or the other tutorials here, please either send a message to the [mailing list](#) or join the chat room at [sleek@conference.jabber.org](mailto:sleek@conference.jabber.org).

---

If you have not yet installed SleekXMPP, do so now by either checking out a version from [Github](#), or installing it using `pip` or `easy_install`.

```
pip install sleekxmpp # Or: easy_install sleekxmpp
```

As a basic starting project, we will create an echo bot which will reply to any messages sent to it. We will also go through adding some basic command line configuration for enabling or disabling debug log outputs and setting the username and password for the bot.

For the command line options processing, we will use the built-in `optparse` module and the `getpass` module for reading in passwords.

### 2.1.1 TL;DR Just Give Me the Code

As you wish: *the completed example*.

### 2.1.2 Overview

To get started, here is a brief outline of the structure that the final project will have:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys
import logging
import getpass
from optparse import OptionParser

import sleekxmpp

'''Here we will create out echo bot class'''

if __name__ == '__main__':
    '''Here we will configure and read command line options'''
```

```
'''Here we will instantiate our echo bot'''

'''Finally, we connect the bot and start listening for messages'''
```

### 2.1.3 Default Encoding

XMPP requires support for UTF-8 and so SleekXMPP must use UTF-8 as well. In Python3 this is simple because Unicode is the default string type. For Python2.6+ the situation is not as easy because standard strings are simply byte arrays and use ASCII. We can get Python to use UTF-8 as the default encoding by including:

```
if sys.version_info < (3, 0):
    from sleekxmpp.util.misc_ops import setdefaultencoding
    setdefaultencoding('utf8')
```

**Warning:** Until we are able to ensure that SleekXMPP will always use Unicode in Python2.6+, this may cause issues embedding SleekXMPP into other applications which assume ASCII encoding.

### 2.1.4 Creating the EchoBot Class

There are three main types of entities within XMPP — servers, components, and clients. Since our echo bot will only be responding to a few people, and won't need to remember thousands of users, we will use a client connection. A client connection is the same type that you use with your standard IM client such as Pidgin or Psi.

SleekXMPP comes with a `ClientXMPP` class which we can extend to add our message echoing feature. `ClientXMPP` requires the parameters `jid` and `password`, so we will let our `EchoBot` class accept those as well.

```
class EchoBot(sleekxmpp.ClientXMPP):

    def __init__(self, jid, password):
        super(EchoBot, self).__init__(jid, password)
```

#### Handling Session Start

The XMPP spec requires clients to broadcast its presence and retrieve its roster (buddy list) once it connects and establishes a session with the XMPP server. Until these two tasks are completed, some servers may not deliver or send messages or presence notifications to the client. So we now need to be sure that we retrieve our roster and send an initial presence once the session has started. To do that, we will register an event handler for the `session_start` event.

```
def __init__(self, jid, password):
    super(EchoBot, self).__init__(jid, password)

    self.add_event_handler('session_start', self.start)
```

Since we want the method `self.start` to execute when the `session_start` event is triggered, we also need to define the `self.start` handler.

```
def start(self, event):
    self.send_presence()
    self.get_roster()
```

**Warning:** Not sending an initial presence and retrieving the roster when using a client instance can prevent your program from receiving presence notifications or messages depending on the XMPP server you have chosen.

Our event handler, like every event handler, accepts a single parameter which typically is the stanza that was received that caused the event. In this case, `event` will just be an empty dictionary since there is no associated data.

Our first task of sending an initial presence is done using `send_presence`. Calling `send_presence` without any arguments will send the simplest stanza allowed in XMPP:

```
<presence />
```

The second requirement is fulfilled using `get_roster`, which will send an IQ stanza requesting the roster to the server and then wait for the response. You may be wondering what `get_roster` returns since we are not saving any return value. The roster data is saved by an internal handler to `self.roster`, and in the case of a `ClientXMPP` instance to `self.client_roster`. (The difference between `self.roster` and `self.client_roster` is that `self.roster` supports storing roster information for multiple JIDs, which is useful for components, whereas `self.client_roster` stores roster data for just the client's JID.)

It is possible for a timeout to occur while waiting for the server to respond, which can happen if the network is excessively slow or the server is no longer responding. In that case, an `IQTimeout` is raised. Similarly, an `IQError` exception can be raised if the request contained bad data or requested the roster for the wrong user. In either case, you can wrap the `get_roster()` call in a `try/except` block to retry the roster retrieval process.

The XMPP stanzas from the roster retrieval process could look like this:

```
<iq type="get">
  <query xmlns="jabber:iq:roster" />
</iq>

<iq type="result" to="echobot@example.com" from="example.com">
  <query xmlns="jabber:iq:roster">
    <item jid="friend@example.com" subscription="both" />
  </query>
</iq>
```

## Responding to Messages

Now that an `EchoBot` instance handles `session_start`, we can begin receiving and responding to messages. Now we can register a handler for the `message` event that is raised whenever a message is received.

```
def __init__(self, jid, password):
    super(EchoBot, self).__init__(jid, password)

    self.add_event_handler('session_start', self.start)
    self.add_event_handler('message', self.message)
```

The `message` event is fired whenever a `<message />` stanza is received, including for group chat messages, errors, etc. Properly responding to messages thus requires checking the `'type'` interface of the message *stanza object*. For responding to only messages addressed to our bot (and not from a chat room), we check that the type is either `normal` or `chat`. (Other potential types are `error`, `headline`, and `groupchat`.)

```
def message(self, msg):
    if msg['type'] in ('normal', 'chat'):
        msg.reply("Thanks for sending:\n%s" % msg['body']).send()
```

Let's take a closer look at the `.reply()` method used above. For message stanzas, `.reply()` accepts the parameter `body` (also as the first positional argument), which is then used as the value of the `<body />` element of the message.

Setting the appropriate `to` JID is also handled by `.reply()`.

Another way to have sent the reply message would be to use `send_message`, which is a convenience method for generating and sending a message based on the values passed to it. If we were to use this method, the above code would look as so:

```
def message(self, msg):
    if msg['type'] in ('normal', 'chat'):
        self.send_message(mto=msg['from'],
                           mbody='Thanks for sending:\n%s' % msg['body'])
```

Whichever method you choose to use, the results in action will look like this:

```
<message to="echobot@example.com" from="someuser@example.net" type="chat">
  <body>Hej!</body>
</message>

<message to="someuser@example.net" type="chat">
  <body>Thanks for sending:
  Hej!</body>
</message>
```

---

**Note:** XMPP does not require stanzas sent by a client to include a `from` attribute, and leaves that responsibility to the XMPP server. However, if a sent stanza does include a `from` attribute, it must match the full JID of the client or some servers will reject it. SleekXMPP thus leaves out the `from` attribute when replying using a client connection.

---

## 2.1.5 Command Line Arguments and Logging

While this isn't part of SleekXMPP itself, we do want our echo bot program to be able to accept a JID and password from the command line instead of hard coding them. We will use the `optparse` module for this, though there are several alternative methods, including the newer `argparse` module.

We want to accept three parameters: the JID for the echo bot, its password, and a flag for displaying the debugging logs. We also want these to be optional parameters, since passing a password directly through the command line can be a security risk.

```
if __name__ == '__main__':
    optp = OptionParser()

    optp.add_option('-d', '--debug', help='set logging to DEBUG',
                    action='store_const', dest='loglevel',
                    const=logging.DEBUG, default=logging.INFO)
    optp.add_option("-j", "--jid", dest="jid",
                    help="JID to use")
    optp.add_option("-p", "--password", dest="password",
                    help="password to use")

    opts, args = optp.parse_args()

    if opts.jid is None:
        opts.jid = raw_input("Username: ")
    if opts.password is None:
        opts.password = getpass.getpass("Password: ")
```

Since we included a flag for enabling debugging logs, we need to configure the logging module to behave accordingly.

```
if __name__ == '__main__':  
  
    # .. option parsing from above ..  
  
    logging.basicConfig(level=opts.loglevel,  
                        format='%(levelname)-8s %(message)s')
```

## 2.1.6 Connecting to the Server and Processing

There are three steps remaining until our echo bot is complete:

1. We need to instantiate the bot.
2. The bot needs to connect to an XMPP server.
3. We have to instruct the bot to start running and processing messages.

Creating the bot is straightforward, but we can also perform some configuration at this stage. For example, let's say we want our bot to support [service discovery](#) and [pings](#):

```
if __name__ == '__main__':  
  
    # .. option parsing and logging steps from above  
  
    xmpp = EchoBot(opts.jid, opts.password)  
    xmpp.register_plugin('xep_0030') # Service Discovery  
    xmpp.register_plugin('xep_0199') # Ping
```

If the `EchoBot` class had a hard dependency on a plugin, we could register that plugin in the `EchoBot.__init__` method instead.

---

**Note:** If you are using the OpenFire server, you will need to include an additional configuration step. OpenFire supports a different version of SSL than what most servers and SleekXMPP support.

```
import ssl  
xmpp.ssl_version = ssl.PROTOCOL_SSLv3
```

---

Now we're ready to connect and begin echoing messages. If you have the package `dnspython` installed, then the `sleekxmpp.clientxmpp.ClientXMPP()` method will perform a DNS query to find the appropriate server to connect to for the given JID. If you do not have `dnspython`, then SleekXMPP will attempt to connect to the hostname used by the JID, unless an address tuple is supplied to `sleekxmpp.clientxmpp.ClientXMPP()`.

```
if __name__ == '__main__':  
  
    # .. option parsing & echo bot configuration  
  
    if xmpp.connect():  
        xmpp.process(block=True)  
    else:  
        print('Unable to connect')
```

---

**Note:** For Google Talk users without `dnspython` installed, the above code should look like:

```
if __name__ == '__main__':  
  
    # .. option parsing & echo bot configuration
```

```
if xmpp.connect(('talk.google.com', 5222)):
    xmpp.process(block=True)
else:
    print('Unable to connect')
```

---

To begin responding to messages, you'll see we called `sleekxmpp.basexmpp.BaseXMPP.process()` which will start the event handling, send queue, and XML reader threads. It will also call the `sleekxmpp.plugins.base.base_plugin.post_init()` method on all registered plugins. By passing `block=True` to `sleekxmpp.basexmpp.BaseXMPP.process()` we are running the main processing loop in the main thread of execution. The `sleekxmpp.basexmpp.BaseXMPP.process()` call will not return until after SleekXMPP disconnects. If you need to run the client in the background for another program, use `block=False` to spawn the processing loop in its own thread.

---

**Note:** Before 1.0, controlling the blocking behaviour of `sleekxmpp.basexmpp.BaseXMPP.process()` was done via the `threaded` argument. This arrangement was a source of confusion because some users interpreted that as controlling whether or not SleekXMPP used threads at all, instead of how the processing loop itself was spawned.

The statements `xmpp.process(threaded=False)` and `xmpp.process(block=True)` are equivalent.

---

### 2.1.7 The Final Product

Here then is what the final result should look like after working through the guide above. The code can also be found in the SleekXMPP [examples directory](#).

You can run the code using:

```
python echobot.py -d -j echobot@example.com
```

which will prompt for the password and then begin echoing messages. To test, open your regular IM client and start a chat with the echo bot. Messages you send to it should be mirrored back to you. Be careful if you are using the same JID for the echo bot that you also have logged in with another IM client. Messages could be routed to your IM client instead of the bot.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    SleekXMPP: The Sleek XMPP Library
    Copyright (C) 2010 Nathanael C. Fritz
    This file is part of SleekXMPP.

    See the file LICENSE for copying permission.
"""

import sys
import logging
import getpass
from optparse import OptionParser

import sleekxmpp

# Python versions before 3.0 do not use UTF-8 encoding
# by default. To ensure that Unicode is handled properly
# throughout SleekXMPP, we will set the default encoding
```

```
# ourselves to UTF-8.
if sys.version_info < (3, 0):
    from sleekxmpp.util.misc_ops import setdefaultencoding
    setdefaultencoding('utf8')
else:
    raw_input = input

class EchoBot(sleekxmpp.ClientXMPP):

    """
    A simple SleekXMPP bot that will echo messages it
    receives, along with a short thank you message.
    """

    def __init__(self, jid, password):
        sleekxmpp.ClientXMPP.__init__(self, jid, password)

        # The session_start event will be triggered when
        # the bot establishes its connection with the server
        # and the XML streams are ready for use. We want to
        # listen for this event so that we we can initialize
        # our roster.
        self.add_event_handler("session_start", self.start)

        # The message event is triggered whenever a message
        # stanza is received. Be aware that that includes
        # MUC messages and error messages.
        self.add_event_handler("message", self.message)

    def start(self, event):
        """
        Process the session_start event.

        Typical actions for the session_start event are
        requesting the roster and broadcasting an initial
        presence stanza.

        Arguments:
            event -- An empty dictionary. The session_start
                    event does not provide any additional
                    data.
        """
        self.send_presence()
        self.get_roster()

    def message(self, msg):
        """
        Process incoming message stanzas. Be aware that this also
        includes MUC messages and error messages. It is usually
        a good idea to check the messages's type before processing
        or sending replies.

        Arguments:
            msg -- The received message stanza. See the documentation
                  for stanza objects and the Message stanza to see
                  how it may be used.
        """
```

```
        if msg['type'] in ('chat', 'normal'):
            msg.reply("Thanks for sending\n%(body)s" % msg).send()

if __name__ == '__main__':
    # Setup the command line arguments.
    optp = OptionParser()

    # Output verbosity options.
    optp.add_option('-q', '--quiet', help='set logging to ERROR',
                    action='store_const', dest='loglevel',
                    const=logging.ERROR, default=logging.INFO)
    optp.add_option('-d', '--debug', help='set logging to DEBUG',
                    action='store_const', dest='loglevel',
                    const=logging.DEBUG, default=logging.INFO)
    optp.add_option('-v', '--verbose', help='set logging to COMM',
                    action='store_const', dest='loglevel',
                    const=5, default=logging.INFO)

    # JID and password options.
    optp.add_option("-j", "--jid", dest="jid",
                    help="JID to use")
    optp.add_option("-p", "--password", dest="password",
                    help="password to use")

    opts, args = optp.parse_args()

    # Setup logging.
    logging.basicConfig(level=opts.loglevel,
                        format='%(levelname)-8s %(message)s')

    if opts.jid is None:
        opts.jid = raw_input("Username: ")
    if opts.password is None:
        opts.password = getpass.getpass("Password: ")

    # Setup the EchoBot and register plugins. Note that while plugins may
    # have interdependencies, the order in which you register them does
    # not matter.
    xmpp = EchoBot(opts.jid, opts.password)
    xmpp.register_plugin('xep_0030') # Service Discovery
    xmpp.register_plugin('xep_0004') # Data Forms
    xmpp.register_plugin('xep_0060') # PubSub
    xmpp.register_plugin('xep_0199') # XMPP Ping

    # If you are connecting to Facebook and wish to use the
    # X-FACEBOOK-PLATFORM authentication mechanism, you will need
    # your API key and an access token. Then you'll set:
    # xmpp.credentials['api_key'] = 'THE_API_KEY'
    # xmpp.credentials['access_token'] = 'THE_ACCESS_TOKEN'

    # If you are connecting to MSN, then you will need an
    # access token, and it does not matter what JID you
    # specify other than that the domain is 'messenger.live.com',
    # so '@messenger.live.com' will work. You can specify
    # the access token as so:
    # xmpp.credentials['access_token'] = 'THE_ACCESS_TOKEN'
```



```

# If you are working with an OpenFire server, you may need
# to adjust the SSL version used:
# xmpp.ssl_version = ssl.PROTOCOL_SSLv3

# If you want to verify the SSL certificates offered by a server:
# xmpp.ca_certs = "path/to/ca/cert"

# Connect to the XMPP server and start processing XMPP stanzas.
if xmpp.connect():
    # If you do not have the dnspython library installed, you will need
    # to manually specify the name of the server if it does not match
    # the one in the JID. For example, to use Google Talk you would
    # need to use:
    #
    # if xmpp.connect(('talk.google.com', 5222)):
    #     ...
    xmpp.process(block=False)
    print("Done")
else:
    print("Unable to connect.")

```

## 2.2 Sign in, Send a Message, and Disconnect

**Note:** If you have any issues working through this quickstart guide or the other tutorials here, please either send a message to the [mailing list](#) or join the chat room at [sleek@conference.jabber.org](mailto:sleek@conference.jabber.org).

A common use case for SleekXMPP is to send one-off messages from time to time. For example, one use case could be sending out a notice when a shell script finishes a task.

We will create our one-shot bot based on the pattern explained in *SleekXMPP Quickstart - Echo Bot*. To start, we create a client class based on `ClientXMPP` and register a handler for the `session_start` event. We will also accept parameters for the JID that will receive our message, and the string content of the message.

```

import sleekxmpp

class SendMsgBot(sleekxmpp.ClientXMPP):

    def __init__(self, jid, password, recipient, msg):
        super(SendMsgBot, self).__init__(jid, password)

        self.recipient = recipient
        self.msg = msg

        self.add_event_handler('session_start', self.start)

    def start(self, event):
        self.send_presence()
        self.get_roster()

```

Note that as in *SleekXMPP Quickstart - Echo Bot*, we need to include send an initial presence and request the roster. Next, we want to send our message, and to do that we will use `send_message`.

```

def start(self, event):
    self.send_presence()

```

```
self.get_roster()

self.send_message(mto=self.recipient, mbody=self.msg)
```

Finally, we need to disconnect the client using `disconnect`. Now, sent stanzas are placed in a queue to pass them to the send thread. If we were to call `disconnect` without any parameters, then it is possible for the client to disconnect before the send queue is processed and the message is actually sent on the wire. To ensure that our message is processed, we use `disconnect(wait=True)`.

```
def start(self, event):
    self.send_presence()
    self.get_roster()

    self.send_message(mto=self.recipient, mbody=self.msg)

    self.disconnect(wait=True)
```

**Warning:** If you happen to be adding stanzas to the send queue faster than the send thread can process them, then `disconnect(wait=True)` will block and not disconnect.

## 2.2.1 Final Product

The final step is to create a small runner script for initialising our `SendMsgBot` class and adding some basic configuration options. By following the basic boilerplate pattern in *SleekXMPP Quickstart - Echo Bot*, we arrive at the code below. To experiment with this example, you can use:

```
python send_client.py -d -j oneshot@example.com -t someone@example.net -m "This is a message"
```

which will prompt for the password and then log in, send your message, and then disconnect. To test, open your regular IM client with the account you wish to send messages to. When you run the `send_client.py` example and instruct it to send your IM client account a message, you should receive the message you gave. If the two JIDs you use also have a mutual presence subscription (they're on each other's buddy lists) then you will also see the `SendMsgBot` client come online and then go offline.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    SleekXMPP: The Sleek XMPP Library
    Copyright (C) 2010 Nathanael C. Fritz
    This file is part of SleekXMPP.

    See the file LICENSE for copying permission.
"""

import sys
import logging
import getpass
from optparse import OptionParser

import sleekxmpp

# Python versions before 3.0 do not use UTF-8 encoding
# by default. To ensure that Unicode is handled properly
# throughout SleekXMPP, we will set the default encoding
# ourselves to UTF-8.
```

---

```

if sys.version_info < (3, 0):
    from sleekxmpp.util.misc_ops import setdefaultencoding
    setdefaultencoding('utf8')
else:
    raw_input = input

class SendMsgBot(sleekxmpp.ClientXMPP):

    """
    A basic SleekXMPP bot that will log in, send a message,
    and then log out.
    """

    def __init__(self, jid, password, recipient, message):
        sleekxmpp.ClientXMPP.__init__(self, jid, password)

        # The message we wish to send, and the JID that
        # will receive it.
        self.recipient = recipient
        self.msg = message

        # The session_start event will be triggered when
        # the bot establishes its connection with the server
        # and the XML streams are ready for use. We want to
        # listen for this event so that we we can initialize
        # our roster.
        self.add_event_handler("session_start", self.start, threaded=True)

    def start(self, event):
        """
        Process the session_start event.

        Typical actions for the session_start event are
        requesting the roster and broadcasting an initial
        presence stanza.

        Arguments:
            event -- An empty dictionary. The session_start
                    event does not provide any additional
                    data.
        """
        self.send_presence()
        self.get_roster()

        self.send_message(mto=self.recipient,
                          mbody=self.msg,
                          mtype='chat')

        # Using wait=True ensures that the send queue will be
        # emptied before ending the session.
        self.disconnect(wait=True)

if __name__ == '__main__':
    # Setup the command line arguments.
    optp = OptionParser()

```

```
# Output verbosity options.
optp.add_option('-q', '--quiet', help='set logging to ERROR',
                action='store_const', dest='loglevel',
                const=logging.ERROR, default=logging.INFO)
optp.add_option('-d', '--debug', help='set logging to DEBUG',
                action='store_const', dest='loglevel',
                const=logging.DEBUG, default=logging.INFO)
optp.add_option('-v', '--verbose', help='set logging to COMM',
                action='store_const', dest='loglevel',
                const=5, default=logging.INFO)

# JID and password options.
optp.add_option("-j", "--jid", dest="jid",
                help="JID to use")
optp.add_option("-p", "--password", dest="password",
                help="password to use")
optp.add_option("-t", "--to", dest="to",
                help="JID to send the message to")
optp.add_option("-m", "--message", dest="message",
                help="message to send")

opts, args = optp.parse_args()

# Setup logging.
logging.basicConfig(level=opts.loglevel,
                    format='%(levelname)-8s %(message)s')

if opts.jid is None:
    opts.jid = raw_input("Username: ")
if opts.password is None:
    opts.password = getpass.getpass("Password: ")
if opts.to is None:
    opts.to = raw_input("Send To: ")
if opts.message is None:
    opts.message = raw_input("Message: ")

# Setup the EchoBot and register plugins. Note that while plugins may
# have interdependencies, the order in which you register them does
# not matter.
xmpp = SendMsgBot(opts.jid, opts.password, opts.to, opts.message)
xmpp.register_plugin('xep_0030') # Service Discovery
xmpp.register_plugin('xep_0199') # XMPP Ping

# If you are working with an OpenFire server, you may need
# to adjust the SSL version used:
# xmpp.ssl_version = ssl.PROTOCOL_SSLv3

# If you want to verify the SSL certificates offered by a server:
# xmpp.ca_certs = "path/to/ca/cert"

# Connect to the XMPP server and start processing XMPP stanzas.
if xmpp.connect():
    # If you do not have the dnspython library installed, you will need
    # to manually specify the name of the server if it does not match
    # the one in the JID. For example, to use Google Talk you would
    # need to use:
    #
    # if xmpp.connect(('talk.google.com', 5222)):
```

```
# ...
xmpp.process(block=True)
print("Done")
else:
    print("Unable to connect.")
```

## 2.3 Create and Run a Server Component

---

**Note:** If you have any issues working through this quickstart guide or the other tutorials here, please either send a message to the [mailing list](#) or join the chat room at [sleek@conference.jabber.org](mailto:sleek@conference.jabber.org).

---

If you have not yet installed SleekXMPP, do so now by either checking out a version from [Github](#), or installing it using `pip` or `easy_install`.

```
pip install sleekxmpp # Or: easy_install sleekxmpp
```

Many XMPP applications eventually graduate to requiring to run as a server component in order to meet scalability requirements. To demonstrate how to turn an XMPP client bot into a component, we'll turn the *echobot* example (*SleekXMPP Quickstart - Echo Bot*) into a component version.

The first difference is that we will add an additional import statement:

```
from sleekxmpp.componentxmpp import ComponentXMPP
```

Likewise, we will change the bot's class definition to match:

```
class EchoComponent(ComponentXMPP):

    def __init__(self, jid, secret, server, port):
        ComponentXMPP.__init__(self, jid, secret, server, port)
```

A component instance requires two extra parameters compared to a client instance: `server` and `port`. These specify the name and port of the XMPP server that will be accepting the component. For example, for a MUC component, the following could be used:

```
muc = ComponentXMPP('muc.sleekxmpp.com', '*****', 'sleekxmpp.com', 5555)
```

---

**Note:** The `server` value is **NOT** derived from the provided JID for the component, unlike with client connections.

---

One difference with the component version is that we do not have to handle the `session_start` event if we don't wish to deal with presence.

The other, main difference with components is that the `'from'` value for every stanza must be explicitly set, since components may send stanzas from multiple JIDs. To do so, the `send_message()` and `send_presence()` accept the parameters `mfrom` and `pfrom`, respectively. For any method that uses `Iq` stanzas, `ifrom` may be used.

### 2.3.1 Final Product

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    SleekXMPP: The Sleek XMPP Library
```

*Copyright (C) 2010 Nathanael C. Fritz  
This file is part of SleekXMPP.*

*See the file LICENSE for copying permission.*

```
"""

import sys
import logging
import getpass
from optparse import OptionParser

import sleekxmpp
from sleekxmpp.componentxmpp import ComponentXMPP

# Python versions before 3.0 do not use UTF-8 encoding
# by default. To ensure that Unicode is handled properly
# throughout SleekXMPP, we will set the default encoding
# ourselves to UTF-8.
if sys.version_info < (3, 0):
    from sleekxmpp.util.misc_ops import setdefaultencoding
    setdefaultencoding('utf8')
else:
    raw_input = input

class EchoComponent(ComponentXMPP):

    """
    A simple SleekXMPP component that echoes messages.
    """

    def __init__(self, jid, secret, server, port):
        ComponentXMPP.__init__(self, jid, secret, server, port)

        # You don't need a session_start handler, but that is
        # where you would broadcast initial presence.

        # The message event is triggered whenever a message
        # stanza is received. Be aware that that includes
        # MUC messages and error messages.
        self.add_event_handler("message", self.message)

    def message(self, msg):
        """
        Process incoming message stanzas. Be aware that this also
        includes MUC messages and error messages. It is usually
        a good idea to check the messages's type before processing
        or sending replies.

        Since a component may send messages from any number of JIDs,
        it is best to always include a from JID.

        Arguments:
            msg -- The received message stanza. See the documentation
                  for stanza objects and the Message stanza to see
                  how it may be used.
        """
        # The reply method will use the messages 'to' JID as the
```

```

        # outgoing reply's 'from' JID.
        msg.reply("Thanks for sending\n%(body)s" % msg).send()

if __name__ == '__main__':
    # Setup the command line arguments.
    optp = OptionParser()

    # Output verbosity options.
    optp.add_option('-q', '--quiet', help='set logging to ERROR',
                    action='store_const', dest='loglevel',
                    const=logging.ERROR, default=logging.INFO)
    optp.add_option('-d', '--debug', help='set logging to DEBUG',
                    action='store_const', dest='loglevel',
                    const=logging.DEBUG, default=logging.INFO)
    optp.add_option('-v', '--verbose', help='set logging to COMM',
                    action='store_const', dest='loglevel',
                    const=5, default=logging.INFO)

    # JID and password options.
    optp.add_option("-j", "--jid", dest="jid",
                    help="JID to use")
    optp.add_option("-p", "--password", dest="password",
                    help="password to use")
    optp.add_option("-s", "--server", dest="server",
                    help="server to connect to")
    optp.add_option("-P", "--port", dest="port",
                    help="port to connect to")

    opts, args = optp.parse_args()

    if opts.jid is None:
        opts.jid = raw_input("Component JID: ")
    if opts.password is None:
        opts.password = getpass.getpass("Password: ")
    if opts.server is None:
        opts.server = raw_input("Server: ")
    if opts.port is None:
        opts.port = int(raw_input("Port: "))

    # Setup logging.
    logging.basicConfig(level=opts.loglevel,
                        format='%(levelname)-8s %(message)s')

    # Setup the EchoComponent and register plugins. Note that while plugins
    # may have interdependencies, the order in which you register them does
    # not matter.
    xmpp = EchoComponent(opts.jid, opts.password, opts.server, opts.port)
    xmpp.registerPlugin('xep_0030') # Service Discovery
    xmpp.registerPlugin('xep_0004') # Data Forms
    xmpp.registerPlugin('xep_0060') # PubSub
    xmpp.registerPlugin('xep_0199') # XMPP Ping

    # Connect to the XMPP server and start processing XMPP stanzas.
    if xmpp.connect():
        xmpp.process(block=True)
        print("Done")
    else:

```

```
print("Unable to connect.")
```

## 2.4 Manage Presence Subscriptions

## 2.5 Multit-User Chat (MUC) Bot

---

**Note:** If you have any issues working through this quickstart guide or the other tutorials here, please either send a message to the [mailing list](#) or join the chat room at [sleek@conference.jabber.org](mailto:sleek@conference.jabber.org).

---

If you have not yet installed SleekXMPP, do so now by either checking out a version from [Github](#), or installing it using `pip` or `easy_install`.

```
pip install sleekxmpp # Or: easy_install sleekxmpp
```

Now that you’ve got the basic gist of using SleekXMPP by following the echobot example (*SleekXMPP Quickstart - Echo Bot*), we can use one of the bundled plugins to create a very popular XMPP starter project: a [Multi-User Chat](#) (MUC) bot. Our bot will login to an XMPP server, join an MUC chat room and “lurk” indefinitely, responding with a generic message to anyone that mentions its nickname. It will also greet members as they join the chat room.

### 2.5.1 Joining The Room

As usual, our code will be based on the pattern explained in *SleekXMPP Quickstart - Echo Bot*. To start, we create an `MUCBot` class based on `ClientXMPP` and which accepts parameters for the JID of the MUC room to join, and the nick that the bot will use inside the chat room. We also register an *event handler* for the *session\_start* event.

```
import sleekxmpp
```

```
class MUCBot(sleekxmpp.ClientXMPP):
```

```
    def __init__(self, jid, password, room, nick):
        sleekxmpp.ClientXMPP.__init__(self, jid, password)

        self.room = room
        self.nick = nick

        self.add_event_handler("session_start", self.start)
```

After initialization, we also need to register the MUC (XEP-0045) plugin so that we can make use of the group chat plugin’s methods and events.

```
xmpp.register_plugin('xep_0045')
```

Finally, we can make our bot join the chat room once an XMPP session has been established:

```
def start(self, event):
    self.get_roster()
    self.send_presence()
    self.plugin['xep_0045'].joinMUC(self.room,
                                    self.nick,
                                    wait=True)
```

Note that as in *SleekXMPP Quickstart - Echo Bot*, we need to include send an initial presence and request the roster. Next, we want to join the group chat, so we call the `joinMUC` method of the MUC plugin.



---

**Note:** The `plugin` attribute is dictionary that maps to instances of plugins that we have previously registered, by their names.

---

## 2.5.2 Adding Functionality

Currently, our bot just sits dormant inside the chat room, but we would like it to respond to two distinct events by issuing a generic message in each case to the chat room. In particular, when a member mentions the bot's nickname inside the chat room, and when a member joins the chat room.

### Responding to Mentions

Whenever a user mentions our bot's nickname in chat, our bot will respond with a generic message resembling “*I heard that, user.*” We do this by examining all of the messages sent inside the chat and looking for the ones which contain the nickname string.

First, we register an event handler for the `groupchat_message` event inside the bot's `__init__` function.

---

**Note:** We do not register a handler for the `message` event in this bot, but if we did, the group chat message would have been sent to both handlers.

---

```
def __init__(self, jid, password, room, nick):
    sleekxmpp.ClientXMPP.__init__(self, jid, password)

    self.room = room
    self.nick = nick

    self.add_event_handler("session_start", self.start)
    self.add_event_handler("groupchat_message", self.muc_message)
```

Then, we can send our generic message whenever the bot's nickname gets mentioned.

**Warning:** Always check that a message is not from yourself, otherwise you will create an infinite loop responding to your own messages.

```
def muc_message(self, msg):
    if msg['mucnick'] != self.nick and self.nick in msg['body']:
        self.send_message(mto=msg['from'].bare,
                          mbody="I heard that, %s." % msg['mucnick'],
                          mtype='groupchat')
```

### Greeting Members

Now we want to greet member whenever they join the group chat. To do this we will use the dynamic `muc::room@server::got_online`<sup>1</sup> event so it's a good idea to register an event handler for it.

---

<sup>1</sup> this is similar to the `got_online` event and is sent by the `xep_0045` plugin whenever a member joins the referenced MUC chat room.

**Note:** The `groupchat_presence` event is triggered whenever a presence stanza is received from any chat room, including any presences you send yourself. To limit event handling to a single room, use the events `muc::room@server::presence`, `muc::room@server::got_online`, or `muc::room@server::got_offline`.

---

```
def __init__(self, jid, password, room, nick):
    sleekxmpp.ClientXMPP.__init__(self, jid, password)

    self.room = room
    self.nick = nick

    self.add_event_handler("session_start", self.start)
    self.add_event_handler("groupchat_message", self.muc_message)
    self.add_event_handler("muc::%s::got_online" % self.room,
                           self.muc_online)
```

Now all that's left to do is to greet them:

```
def muc_online(self, presence):
    if presence['muc']['nick'] != self.nick:
        self.send_message(mto=presence['from'].bare,
                          mbody="Hello, %s %s" % (presence['muc']['role'],
                                                  presence['muc']['nick']),
                          mtype='groupchat')
```

## 2.5.3 Final Product

The final step is to create a small runner script for initialising our `MUCBot` class and adding some basic configuration options. By following the basic boilerplate pattern in *SleekXMPP Quickstart - Echo Bot*, we arrive at the code below. To experiment with this example, you can use:

```
python muc.py -d -j jid@example.com -r room@muc.example.net -n lurkbot
```

which will prompt for the password, log in, and join the group chat. To test, open your regular IM client and join the same group chat that you sent the bot to. You will see `lurkbot` as one of the members in the group chat, and that it greeted you upon entry. Send a message with the string “lurkbot” inside the body text, and you will also see that it responds with our pre-programmed customized message.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    SleekXMPP: The Sleek XMPP Library
    Copyright (C) 2010 Nathanael C. Fritz
    This file is part of SleekXMPP.

    See the file LICENSE for copying permission.
"""

import sys
import logging
import getpass
from optparse import OptionParser

import sleekxmpp

# Python versions before 3.0 do not use UTF-8 encoding
```

```

# by default. To ensure that Unicode is handled properly
# throughout SleekXMPP, we will set the default encoding
# ourselves to UTF-8.
if sys.version_info < (3, 0):
    from sleekxmpp.util.misc_ops import setdefaultencoding
    setdefaultencoding('utf8')
else:
    raw_input = input

class MUCBot(sleekxmpp.ClientXMPP):

    """
    A simple SleekXMPP bot that will greets those
    who enter the room, and acknowledge any messages
    that mentions the bot's nickname.
    """

    def __init__(self, jid, password, room, nick):
        sleekxmpp.ClientXMPP.__init__(self, jid, password)

        self.room = room
        self.nick = nick

        # The session_start event will be triggered when
        # the bot establishes its connection with the server
        # and the XML streams are ready for use. We want to
        # listen for this event so that we we can initialize
        # our roster.
        self.add_event_handler("session_start", self.start)

        # The groupchat_message event is triggered whenever a message
        # stanza is received from any chat room. If you also also
        # register a handler for the 'message' event, MUC messages
        # will be processed by both handlers.
        self.add_event_handler("groupchat_message", self.muc_message)

        # The groupchat_presence event is triggered whenever a
        # presence stanza is received from any chat room, including
        # any presences you send yourself. To limit event handling
        # to a single room, use the events muc::room@server::presence,
        # muc::room@server::got_online, or muc::room@server::got_offline.
        self.add_event_handler("muc::%s::got_online" % self.room,
                               self.muc_online)

    def start(self, event):
        """
        Process the session_start event.

        Typical actions for the session_start event are
        requesting the roster and broadcasting an initial
        presence stanza.

        Arguments:
            event -- An empty dictionary. The session_start
                     event does not provide any additional
                     data.

```

```
"""
self.get_roster()
self.send_presence()
self.plugin['xep_0045'].joinMUC(self.room,
                                self.nick,
                                # If a room password is needed, use:
                                # password=the_room_password,
                                wait=True)

def muc_message(self, msg):
    """
    Process incoming message stanzas from any chat room. Be aware
    that if you also have any handlers for the 'message' event,
    message stanzas may be processed by both handlers, so check
    the 'type' attribute when using a 'message' event handler.

    Whenever the bot's nickname is mentioned, respond to
    the message.

    IMPORTANT: Always check that a message is not from yourself,
    otherwise you will create an infinite loop responding
    to your own messages.

    This handler will reply to messages that mention
    the bot's nickname.

    Arguments:
        msg -- The received message stanza. See the documentation
               for stanza objects and the Message stanza to see
               how it may be used.
    """
    if msg['mucnick'] != self.nick and self.nick in msg['body']:
        self.send_message(mto=msg['from'].bare,
                          mbody="I heard that, %s." % msg['mucnick'],
                          mtype='groupchat')

def muc_online(self, presence):
    """
    Process a presence stanza from a chat room. In this case,
    presences from users that have just come online are
    handled by sending a welcome message that includes
    the user's nickname and role in the room.

    Arguments:
        presence -- The received presence stanza. See the
                    documentation for the Presence stanza
                    to see how else it may be used.
    """
    if presence['muc']['nick'] != self.nick:
        self.send_message(mto=presence['from'].bare,
                          mbody="Hello, %s %s" % (presence['muc']['role'],
                                                  presence['muc']['nick']),
                          mtype='groupchat')

if __name__ == '__main__':
    # Setup the command line arguments.
    optp = OptionParser()
```

```

# Output verbosity options.
optp.add_option('-q', '--quiet', help='set logging to ERROR',
                action='store_const', dest='loglevel',
                const=logging.ERROR, default=logging.INFO)
optp.add_option('-d', '--debug', help='set logging to DEBUG',
                action='store_const', dest='loglevel',
                const=logging.DEBUG, default=logging.INFO)
optp.add_option('-v', '--verbose', help='set logging to COMM',
                action='store_const', dest='loglevel',
                const=5, default=logging.INFO)

# JID and password options.
optp.add_option("-j", "--jid", dest="jid",
                help="JID to use")
optp.add_option("-p", "--password", dest="password",
                help="password to use")
optp.add_option("-r", "--room", dest="room",
                help="MUC room to join")
optp.add_option("-n", "--nick", dest="nick",
                help="MUC nickname")

opts, args = optp.parse_args()

# Setup logging.
logging.basicConfig(level=opts.loglevel,
                    format='%(levelname)-8s %(message)s')

if opts.jid is None:
    opts.jid = raw_input("Username: ")
if opts.password is None:
    opts.password = getpass.getpass("Password: ")
if opts.room is None:
    opts.room = raw_input("MUC room: ")
if opts.nick is None:
    opts.nick = raw_input("MUC nickname: ")

# Setup the MUCBot and register plugins. Note that while plugins may
# have interdependencies, the order in which you register them does
# not matter.
xmpp = MUCBot(opts.jid, opts.password, opts.room, opts.nick)
xmpp.register_plugin('xep_0030') # Service Discovery
xmpp.register_plugin('xep_0045') # Multi-User Chat
xmpp.register_plugin('xep_0199') # XMPP Ping

# Connect to the XMPP server and start processing XMPP stanzas.
if xmpp.connect():
    # If you do not have the dnspython library installed, you will need
    # to manually specify the name of the server if it does not match
    # the one in the JID. For example, to use Google Talk you would
    # need to use:
    #
    # if xmpp.connect(('talk.google.com', 5222)):
    #     ...
    xmpp.process(block=True)
    print("Done")
else:
    print("Unable to connect.")

```

## 2.6 Enable HTTP Proxy Support

---

**Note:** If you have any issues working through this quickstart guide or the other tutorials here, please either send a message to the [mailing list](#) or join the chat room at [sleek@conference.jabber.org](mailto:sleek@conference.jabber.org).

---

In some instances, you may wish to route XMPP traffic through an HTTP proxy, probably to get around restrictive firewalls. SleekXMPP provides support for basic HTTP proxying with DIGEST authentication.

Enabling proxy support is done in two steps. The first is to instruct SleekXMPP to use a proxy, and the second is to configure the proxy details:

```
xmpp = ClientXMPP(...)
xmpp.use_proxy = True
xmpp.proxy_config = {
    'host': 'proxy.example.com',
    'port': 5555,
    'username': 'example_user',
    'password': '*****'
}
```

The 'username' and 'password' fields are optional if the proxy does not require authentication.

### 2.6.1 The Final Product

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    SleekXMPP: The Sleek XMPP Library
    Copyright (C) 2010 Nathanael C. Fritz
    This file is part of SleekXMPP.

    See the file LICENSE for copying permission.
"""

import sys
import logging
import getpass
from optparse import OptionParser

import sleekxmpp

# Python versions before 3.0 do not use UTF-8 encoding
# by default. To ensure that Unicode is handled properly
# throughout SleekXMPP, we will set the default encoding
# ourselves to UTF-8.
if sys.version_info < (3, 0):
    from sleekxmpp.util.misc_ops import setdefaultencoding
    setdefaultencoding('utf8')
else:
    raw_input = input

class EchoBot(sleekxmpp.ClientXMPP):
```

```

"""
A simple SleekXMPP bot that will echo messages it
receives, along with a short thank you message.
"""

def __init__(self, jid, password):
    sleekxmpp.ClientXMPP.__init__(self, jid, password)

    # The session_start event will be triggered when
    # the bot establishes its connection with the server
    # and the XML streams are ready for use. We want to
    # listen for this event so that we can initialize
    # our roster.
    self.add_event_handler("session_start", self.start)

    # The message event is triggered whenever a message
    # stanza is received. Be aware that that includes
    # MUC messages and error messages.
    self.add_event_handler("message", self.message)

def start(self, event):
    """
    Process the session_start event.

    Typical actions for the session_start event are
    requesting the roster and broadcasting an initial
    presence stanza.

    Arguments:
        event -- An empty dictionary. The session_start
        event does not provide any additional
        data.
    """
    self.send_presence()
    self.get_roster()

def message(self, msg):
    """
    Process incoming message stanzas. Be aware that this also
    includes MUC messages and error messages. It is usually
    a good idea to check the messages's type before processing
    or sending replies.

    Arguments:
        msg -- The received message stanza. See the documentation
        for stanza objects and the Message stanza to see
        how it may be used.
    """
    msg.reply("Thanks for sending\n%(body)s" % msg).send()

if __name__ == '__main__':
    # Setup the command line arguments.
    optp = OptionParser()

    # Output verbosity options.
    optp.add_option('-q', '--quiet', help='set logging to ERROR',
                    action='store_const', dest='loglevel',

```

```
        const=logging.ERROR, default=logging.INFO)
optp.add_option('-d', '--debug', help='set logging to DEBUG',
               action='store_const', dest='loglevel',
               const=logging.DEBUG, default=logging.INFO)
optp.add_option('-v', '--verbose', help='set logging to COMM',
               action='store_const', dest='loglevel',
               const=5, default=logging.INFO)

# JID and password options.
optp.add_option("-j", "--jid", dest="jid",
               help="JID to use")
optp.add_option("-p", "--password", dest="password",
               help="password to use")
optp.add_option("--phost", dest="proxy_host",
               help="Proxy hostname")
optp.add_option("--pport", dest="proxy_port",
               help="Proxy port")
optp.add_option("--puser", dest="proxy_user",
               help="Proxy username")
optp.add_option("--ppass", dest="proxy_pass",
               help="Proxy password")

opts, args = optp.parse_args()

# Setup logging.
logging.basicConfig(level=opts.loglevel,
                   format='%(levelname)-8s %(message)s')

if opts.jid is None:
    opts.jid = raw_input("Username: ")
if opts.password is None:
    opts.password = getpass.getpass("Password: ")
if opts.proxy_host is None:
    opts.proxy_host = raw_input("Proxy host: ")
if opts.proxy_port is None:
    opts.proxy_port = raw_input("Proxy port: ")
if opts.proxy_user is None:
    opts.proxy_user = raw_input("Proxy username: ")
if opts.proxy_pass is None and opts.proxy_user:
    opts.proxy_pass = getpass.getpass("Proxy password: ")

# Setup the EchoBot and register plugins. Note that while plugins may
# have interdependencies, the order in which you register them does
# not matter.
xmpp = EchoBot(opts.jid, opts.password)
xmpp.register_plugin('xep_0030') # Service Discovery
xmpp.register_plugin('xep_0004') # Data Forms
xmpp.register_plugin('xep_0060') # PubSub
xmpp.register_plugin('xep_0199') # XMPP Ping

# If you are working with an OpenFire server, you may need
# to adjust the SSL version used:
xmpp.ssl_version = ssl.PROTOCOL_SSLv3

# If you want to verify the SSL certificates offered by a server:
xmpp.ca_certs = "path/to/ca/cert"
```



```

xmpp.use_proxy = True
xmpp.proxy_config = {
    'host': opts.proxy_host,
    'port': int(opts.proxy_port),
    'username': opts.proxy_user,
    'password': opts.proxy_pass}

# Connect to the XMPP server and start processing XMPP stanzas.
if xmpp.connect():
    # If you do not have the dnspython library installed, you will need
    # to manually specify the name of the server if it does not match
    # the one in the JID. For example, to use Google Talk you would
    # need to use:
    #
    # if xmpp.connect(('talk.google.com', 5222)):
    #     ...
    xmpp.process(block=True)
    print("Done")
else:
    print("Unable to connect.")

```

## 2.7 Send a Message Every 5 Minutes

## 2.8 Send/Receive IQ Stanzas

Unlike Message and Presence stanzas which only use text data for basic usage, Iq stanzas require using XML payloads, and generally entail creating a new SleekXMPP plugin to provide the necessary convenience methods to make working with them easier.

### 2.8.1 Basic Use

XMPP's use of Iq stanzas is built around namespaced `<query />` elements. For clients, just sending the empty `<query />` element will suffice for retrieving information. For example, a very basic implementation of service discovery would just need to be able to send:

```

<iq to="user@example.com" type="get" id="1">
  <query xmlns="http://jabber.org/protocol/disco#info" />
</iq>

```

### Creating Iq Stanzas

SleekXMPP provides built-in support for creating basic Iq stanzas this way. The relevant methods are:

- `make_iq()`
- `make_iq_get()`
- `make_iq_set()`
- `make_iq_result()`
- `make_iq_error()`
- `make_iq_query()`

These methods all follow the same pattern: create or modify an existing Iq stanza, set the 'type' value based on the method name, and finally add a <query /> element with the given namespace. For example, to produce the query above, you would use:

```
self.make_iq_get(queryxmlns='http://jabber.org/protocol/disco#info',
                 ito='user@example.com')
```

### Sending Iq Stanzas

Once an Iq stanza is created, sending it over the wire is done using its `send()` method, like any other stanza object. However, there are a few extra options to control how to wait for the query's response.

These options are:

- `block`: The default behaviour is that `send()` will block until a response is received and the response stanza will be the return value. Setting `block` to `False` will cause the call to return immediately. In which case, you will need to arrange some way to capture the response stanza if you need it.
- `timeout`: When using the blocking behaviour, the call will eventually timeout with an error. The default timeout is 30 seconds, but this may be overridden two ways. To change the timeout globally, set:

```
self.response_timeout = 10
```

To change the timeout for a single call, the `timeout` parameter works:

```
iq.send(timeout=60)
```

- `callback`: When not using a blocking call, using the `callback` argument is a simple way to register a handler that will execute whenever a response is finally received. Using this method, there is no timeout limit. In case you need to remove the callback, the name of the newly created callback is returned.

```
cb_name = iq.send(callback=self.a_callback)
```

```
# ... later if we need to cancel
self.remove_handler(cb_name)
```

Properly working with Iq stanzas requires handling the intended, normal flow, error responses, and timed out requests. To make this easier, two exceptions may be thrown by `send()`: `IqError` and `IqTimeout`. These exceptions only apply to the default, blocking calls.

```
try:
    resp = iq.send()
    # ... do stuff with expected Iq result
except IqError as e:
    err_resp = e.iq
    # ... handle error case
except IqTimeout:
    # ... no response received in time
    pass
```

If you do not care to distinguish between errors and timeouts, then you can combine both cases with a generic `XMPPError` exception:

```
try:
    resp = iq.send()
except XMPPError:
    # ... Don't care about the response
    pass
```

## 2.8.2 Advanced Use

Going beyond the basics provided by SleekXMPP requires building at least a rudimentary SleekXMPP plugin to create a *stanza object* for interfacing with the Iq payload.

**See Also:**

- *Creating a SleekXMPP Plugin*
- *How to Work with Stanza Objects*
- *Using Stream Handlers and Matchers*

The typical way to respond to Iq requests is to register stream handlers. As an example, suppose we create a stanza class named CustomXEP which uses the XML element `<query xmlns="custom-xep" />`, and has a `plugin_attrib` value of `custom_xep`.

There are two types of incoming Iq requests: `get` and `set`. You can register a handler that will accept both and then filter by type as needed, as so:

```
self.register_handler(Callback(
    'CustomXEP Handler',
    StanzaPath('iq/custom_xep'),
    self._handle_custom_iq))

# ...

def _handle_custom_iq(self, iq):
    if iq['type'] == 'get':
        # ...
        pass
    elif iq['type'] == 'set':
        # ...
        pass
    else:
        # ... This will capture error responses too
        pass
```

If you want to filter out query types beforehand, you can adjust the matching filter by using `@type=get` or `@type=set` if you are using the recommended `StanzaPath` matcher.

```
self.register_handler(Callback(
    'CustomXEP Handler',
    StanzaPath('iq@type=get/custom_xep'),
    self._handle_custom_iq_get))

# ...

def _handle_custom_iq_get(self, iq):
    assert(iq['type'] == 'get')
```



# TUTORIALS, FAQs, AND HOW TO GUIDES

## 3.1 Supported XEPS

XEP	Description	Notes
0004	Data forms	
0009	Jabber RPC	
0012	Last Activity	
0030	Service Discovery	
0033	Extended Stanza Addressing	
0045	Multi-User Chat (MUC)	Client-side only
0050	Ad-hoc Commands	
0059	Result Set Management	
0060	Publish/Subscribe (PubSub)	Client-side only
0066	Out-of-band Data	
0078	Non-SASL Authentication	
0082	XMPP Date and Time Profiles	
0085	Chat-State Notifications	
0086	Error Condition Mappings	
0092	Software Version	
0128	Service Discovery Extensions	
0202	Entity Time	
0203	Delayed Delivery	
0224	Attention	
0249	Direct MUC Invitations	

## 3.2 Following *XMPP: The Definitive Guide*

SleekXMPP was featured in the first edition of the O'Reilly book *XMPP: The Definitive Guide* by Peter Saint-Andre, Kevin Smith, and Remko Tronçon. The original source code for the book's examples can be found at <http://github.com/remko/xmpp-tdg>. An updated version of the source code, maintained to stay current with the latest SleekXMPP release, is available at <http://github.com/legastero/xmpp-tdg>.

However, since publication, SleekXMPP has advanced from version 0.2.1 to version 1.0 and there have been several major API changes. The most notable is the introduction of *stanza objects* which have simplified and standardized interactions with the XMPP XML stream.

What follows is a walk-through of *The Definitive Guide* highlighting the changes needed to make the code examples work with version 1.0 of SleekXMPP. These changes have been kept to a minimum to preserve the correlation with the book's explanations, so be aware that some code may not use current best practices.

### 3.2.1 Example 2-2. (Page 26)

#### Implementation of a basic bot that echoes all incoming messages back to its sender.

The echo bot example requires a change to the `handleIncomingMessage` method to reflect the use of the Message *stanza object*. The "jid" field of the message object should now be "from" to match the from attribute of the actual XML message stanza. Likewise, "message" changes to "body" to match the body element of the message stanza.

#### Updated Code

```
def handleIncomingMessage(self, message):
    self.xmpp.sendMessage(message["from"], message["body"])
```

[View full source](#) | [View original code](#)

### 3.2.2 Example 14-1. (Page 215)

#### CheshiR IM bot implementation.

The main event handling method in the Bot class is meant to process both message events and presence update events. With the new changes in SleekXMPP 1.0, extracting a CheshiR status "message" from both types of stanzas requires accessing different attributes. In the case of a message stanza, the "body" attribute would contain the CheshiR message. For a presence event, the information is stored in the "status" attribute. To handle both cases, we can test the type of the given event object and look up the proper attribute based on the type.

Like in the EchoBot example, the expression `event["jid"]` needs to change to `event["from"]` in order to get a JID object for the stanza's sender. Because other functions in CheshiR assume that the JID is a string, the `jid` attribute is used to access the string version of the JID. A check is also added in case `user` is `None`, but the check could (and probably should) be placed in `addMessageFromUser`.

Another change is needed in `handleMessageAddedToBackend` where an HTML-IM response is created. The HTML content should be enclosed in a single element, such as a `<p>` tag.

#### Updated Code

```
def handleIncomingXMPPEvent(self, event):
    msgLocations = {sleekxmpp.stanza.presence.Presence: "status",
                    sleekxmpp.stanza.message.Message: "body"}

    message = event[msgLocations[type(event)]]
    user = self.backend.getUserFromJID(event["from"].jid)
    if user is not None:
        self.backend.addMessageFromUser(message, user)

def handleMessageAddedToBackend(self, message):
    body = message.user + ": " + message.text
    htmlBody = "<p><a href='%s'>%s</a>: %s</p>" % {
        "uri": self.url + "/" + message.user,
        "user": message.user, "message": message.text }
    self.backend.addMessageFromUser(htmlBody, message.user)
```

```
for subscriberJID in self.backend.getSubscriberJIDs(message.user) :
    self.xmpp.sendMessage(subscriberJID, body, mhtml=htmlBody)
```

[View full source](#) | [View original code](#)

### 3.2.3 Example 14-3. (Page 217)

#### Configurable CheshiR IM bot implementation.

**Note:** Since the CheshiR examples build on each other, see previous sections for corrections to code that is not marked as new in the book example.

The main difference for the configurable IM bot is the handling for the data form in `handleConfigurationCommand`. The test for equality with the string "1" is no longer required; SleekXMPP converts boolean data form fields to the values `True` and `False` automatically.

For the method `handleIncomingXMPPPresence`, the attribute "jid" is again converted to "from" to get a JID object for the presence stanza's sender, and the `jid` attribute is used to access the string version of that JID object. A check is also added in case `user` is `None`, but the check could (and probably should) be placed in `getShouldMonitorPresenceFromUser`.

#### Updated Code

```
def handleConfigurationCommand(self, form, sessionId):
    values = form.getValues()
    monitorPresence = values["monitorPresence"]
    jid = self.xmpp.plugin["xep_0050"].sessions[sessionId]["jid"]
    user = self.backend.getUserFromJID(jid)
    self.backend.setShouldMonitorPresenceFromUser(user, monitorPresence)

def handleIncomingXMPPPresence(self, event):
    user = self.backend.getUserFromJID(event["from"].jid)
    if user is not None:
        if self.backend.getShouldMonitorPresenceFromUser(user):
            self.handleIncomingXMPPEvent(event)
```

[View full source](#) | [View original code](#)

### 3.2.4 Example 14-4. (Page 220)

#### CheshiR IM server component implementation.

**Note:** Since the CheshiR examples build on each other, see previous sections for corrections to code that is not marked as new in the book example.

Like several previous examples, a needed change is to replace `subscription["from"]` with `subscription["from"].jid` because the BaseXMPP method `makePresence` requires the JID to be a string.

A correction needs to be made in `handleXMPPPresenceProbe` because a line was left out of the original implementation; the variable `user` is undefined. The JID of the user can be extracted from the presence stanza's `from` attribute.

Since this implementation of CheshiR uses an XMPP component, it must include a `from` attribute in all messages that it sends. Adding the `from` attribute is done by including `mfrom=self.xmpp.jid` in calls to `self.xmpp.sendMessage`.

### Updated Code

```
def handleXMPPPresenceProbe(self, event) :
    self.xmpp.sendPresence(pto = event["from"])

def handleXMPPPresenceSubscription(self, subscription) :
    if subscription["type"] == "subscribe" :
        userJID = subscription["from"].jid
        self.xmpp.sendPresenceSubscription(pto=userJID, ptype="subscribed")
        self.xmpp.sendPresence(pto = userJID)
        self.xmpp.sendPresenceSubscription(pto=userJID, ptype="subscribe")

def handleMessageAddedToBackend(self, message) :
    body = message.user + ": " + message.text
    for subscriberJID in self.backend.getSubscriberJIDs(message.user) :
        self.xmpp.sendMessage(subscriberJID, body, mfrom=self.xmpp.jid)
```

[View full source](#) | [View original code](#)

## 3.2.5 Example 14-6. (Page 223)

### CheshiR IM server component with in-band registration support.

---

**Note:** Since the CheshiR examples build on each other, see previous sections for corrections to code that is not marked as new in the book example.

---

After applying the changes from Example 14-4 above, the registrable component implementation should work correctly.

---

**Tip:** To see how to implement in-band registration as a SleekXMPP plugin, see the tutorial *tutorial-create-plugin*.

---

[View full source](#) | [View original code](#)

## 3.2.6 Example 14-7. (Page 225)

### Extended CheshiR IM server component implementation.

---

**Note:** Since the CheshiR examples build on each other, see previous sections for corrections to code that is not marked as new in the book example.

---

While the final code example can look daunting with all of the changes made, it requires very few modifications to work with the latest version of SleekXMPP. Most differences are the result of CheshiR's backend functions expecting JIDs to be strings so that they can be stripped to bare JIDs. To resolve these, use the `jid` attribute of the JID objects. Also, references to `"message"` and `"jid"` attributes need to be changed to either `"body"` or `"status"`, and either `"from"` or `"to"` depending on if the object is a message or presence stanza and which of the JIDs from the stanza is needed.



## Updated Code

```
def handleIncomingXMPPMessage(self, event) :
    message = self.addRecipientToMessage(event["body"], event["to"].jid)
    user = self.backend.getUserFromJID(event["from"].jid)
    self.backend.addMessageFromUser(message, user)

def handleIncomingXMPPPresence(self, event) :
    if event["to"].jid == self.componentDomain :
        user = self.backend.getUserFromJID(event["from"].jid)
        self.backend.addMessageFromUser(event["status"], user)

...

def handleXMPPPresenceSubscription(self, subscription) :
    if subscription["type"] == "subscribe" :
        userJID = subscription["from"].jid
        user = self.backend.getUserFromJID(userJID)
        contactJID = subscription["to"]
        self.xmpp.sendPresenceSubscription(
            pfrom=contactJID, pto=userJID, ptype="subscribed", pnick=user)
        self.sendPresenceOfContactToUser(contactJID=contactJID, userJID=userJID)
    if contactJID == self.componentDomain :
        self.sendAllContactSubscriptionRequestsToUser(userJID)
```

[View full source](#) | [View original code](#)

## 3.3 How to Work with Stanza Objects

### 3.3.1 Defining Stanza Interfaces

### 3.3.2 Creating Stanza Plugins

### 3.3.3 Creating a Stanza Extension

### 3.3.4 Overriding a Parent Stanza

## 3.4 Creating a SleekXMPP Plugin

One of the goals of SleekXMPP is to provide support for every draft or final XMPP extension (XEP). To do this, SleekXMPP has a plugin mechanism for adding the functionalities required by each XEP. But even though plugins were made to quickly implement and prototype the official XMPP extensions, there is no reason you can't create your own plugin to implement your own custom XMPP-based protocol.

This guide will help walk you through the steps to implement a rudimentary version of [XEP-0077 In-band Registration](#). In-band registration was implemented in example 14-6 (page 223) of [XMPP: The Definitive Guide](#) because there was no SleekXMPP plugin for XEP-0077 at the time of writing. We will partially fix that issue here by turning the example implementation from *XMPP: The Definitive Guide* into a plugin. Again, note that this will not a complete implementation, and a different, more robust, official plugin for XEP-0077 may be added to SleekXMPP in the future.

---

**Note:** The example plugin created in this guide is for the server side of the registration process only. It will **NOT** be able to register new accounts on an XMPP server.

---

### 3.4.1 First Steps

Every plugin inherits from the class `base_plugin`, and must include a `plugin_init` method. While the plugins distributed with SleekXMPP must be placed in the `plugins` directory `sleekxmpp/plugins` to be loaded, custom plugins may be loaded from any module. To do so, use the following form when registering the plugin:

```
self.register_plugin('myplugin', module=mod_containing_my_plugin)
```

The plugin name must be the same as the plugin's class name.

Now, we can open our favorite text editors and create `xep_0077.py` in `SleekXMPP/sleekxmpp/plugins`. We want to do some basic house-keeping and declare the name and description of the XEP we are implementing. If you are creating your own custom plugin, you don't need to include the `xep` attribute.

```
"""
Creating a SleekXMPP Plugin

This is a minimal implementation of XEP-0077 to serve
as a tutorial for creating SleekXMPP plugins.
"""
```

```
from sleekxmpp.plugins.base import base_plugin
```

```
class xep_0077(base_plugin):
    """
    XEP-0077 In-Band Registration
    """

    def plugin_init(self):
        self.description = "In-Band Registration"
        self.xep = "0077"
```

Now that we have a basic plugin, we need to edit `sleekxmpp/plugins/__init__.py` to include our new plugin by adding `'xep_0077'` to the `__all__` declaration.

### 3.4.2 Interacting with Other Plugins

In-band registration is a feature that should be advertised through [Service Discovery](#). To do that, we tell the `xep_0030` plugin to add the `"jabber:iq:register"` feature. We put this call in a method named `post_init` which will be called once the plugin has been loaded; by doing so we advertise that we can do registrations only after we finish activating the plugin.

The `post_init` method needs to call `base_plugin.post_init(self)` which will mark that `post_init` has been called for the plugin. Once the SleekXMPP object begins processing, `post_init` will be called on any plugins that have not already run `post_init`. This allows you to register plugins and their dependencies without needing to worry about the order in which you do so.

**Note:** by adding this call we have introduced a dependency on the XEP-0030 plugin. Be sure to register `'xep_0030'` as well as `'xep_0077'`. SleekXMPP does not automatically load plugin dependencies for you.

```
def post_init(self):
    base_plugin.post_init(self)
    self.xmpp['xep_0030'].add_feature("jabber:iq:register")
```

### 3.4.3 Creating Custom Stanza Objects

Now, the IQ stanzas needed to implement our version of XEP-0077 are not very complex, and we could just interact with the XML objects directly just like in the *XMPP: The Definitive Guide* example. However, creating custom stanza objects is good practice.

We will create a new `Registration` stanza. Following the *XMPP: The Definitive Guide* example, we will add support for a username and password field. We also need two flags: `registered` and `remove`. The `registered` flag is sent when an already registered user attempts to register, along with their registration data. The `remove` flag is a request to unregister a user's account.

Adding additional fields specified in XEP-0077 will not be difficult and is left as an exercise for the reader.

Our `Registration` class needs to start with a few descriptions of its behaviour:

- **namespace** The namespace our stanza object lives in. In this case, `"jabber:iq:register"`.
- **name** The name of the root XML element. In this case, the `query` element.
- **plugin\_attrib** The name to access this type of stanza. In particular, given a registration stanza, the `Registration` object can be found using: `iq_object['register']`.
- **interfaces** A list of dictionary-like keys that can be used with the stanza object. When using "key", if there exists a method of the form `getKey`, `setKey`, or `delKey` (depending on context) then the result of calling that method will be returned. Otherwise, the value of the attribute `key` of the main stanza element is returned if one exists.

**Note:** The accessor methods currently use title case, and not camel case. Thus if you need to access an item named `"methodName"` you will need to use `getMethodname`. This naming convention might change to full camel case in a future version of SleekXMPP.

- **sub\_interfaces** A subset of `interfaces`, but these keys map to the text of any subelements that are direct children of the main stanza element. Thus, referencing `iq_object['register']['username']` will either execute `getUsername` or return the value in the `username` element of the query.

If you need to access an element, say `elem`, that is not a direct child of the main stanza element, you will need to add `getElem`, `setElem`, and `delElem`. See the note above about naming conventions.

```
from sleekxmpp.xmlstream import ElementBase, ET, JID, register_stanza_plugin
from sleekxmpp import Iq
```

```
class Registration(ElementBase):
    namespace = 'jabber:iq:register'
    name = 'query'
    plugin_attrib = 'register'
    interfaces = set(('username', 'password', 'registered', 'remove'))
    sub_interfaces = interfaces

    def getRegistered(self):
        present = self.xml.find('{%s}registered' % self.namespace)
        return present is not None

    def getRemove(self):
        present = self.xml.find('{%s}remove' % self.namespace)
        return present is not None

    def setRegistered(self, registered):
        if registered:
            self.addField('registered')
        else:
```

```
del self['registered']

def setRemove(self, remove):
    if remove:
        self.addField('remove')
    else:
        del self['remove']

def addField(self, name):
    itemXML = ET.Element('{%s}%s' % (self.namespace, name))
    self.xml.append(itemXML)
```

Setting a `sub_interface` attribute to "" will remove that subelement. Since we want to include empty registration fields in our form, we need the `addField` method to add the empty elements.

Since the `registered` and `remove` elements are just flags, we need to add custom logic to enforce the binary behavior.

### 3.4.4 Extracting Stanzas from the XML Stream

Now that we have a custom stanza object, we need to be able to detect when we receive one. To do this, we register a stream handler that will pattern match stanzas off of the XML stream against our stanza object's element name and namespace. To do so, we need to create a `Callback` object which contains an XML fragment that can identify our stanza type. We can add this handler registration to our `plugin_init` method.

Also, we need to associate our `Registration` class with IQ stanzas; that requires the use of the `register_stanza_plugin` function (in `sleekxmpp.xmlstream.stanzabase`) which takes the class of a parent stanza type followed by the `substanza` type. In our case, the parent stanza is an IQ stanza, and the `substanza` is our registration query.

The `__handleRegistration` method referenced in the callback will be our handler function to process registration requests.

```
def plugin_init(self):
    self.description = "In-Band Registration"
    self.xep = "0077"

    self.xmpp.register_handler(
        Callback('In-Band Registration',
            MatchXPath('{%s}iq/{jabber:iq:register}query' % self.xmpp.default_ns),
            self.__handleRegistration))
    register_stanza_plugin(Iq, Registration)
```

### 3.4.5 Handling Incoming Stanzas and Triggering Events

There are six situations that we need to handle to finish our implementation of XEP-0077.

#### Registration Form Request from a New User:

```
<iq type="result">
  <query xmlns="jabber:iq:register">
    <username />
    <password />
  </query>
</iq>
```

#### Registration Form Request from an Existing User:

```
<iq type="result">
  <query xmlns="jabber:iq:register">
    <registered />
    <username>Foo</username>
    <password>hunter2</password>
  </query>
</iq>
```

**Unregister Account:**

```
<iq type="result">
  <query xmlns="jabber:iq:register" />
</iq>
```

**Incomplete Registration:**

```
<iq type="error">
  <query xmlns="jabber:iq:register">
    <username>Foo</username>
  </query>
  <error code="406" type="modify">
    <not-acceptable xmlns="urn:ietf:params:xml:ns:xmpp-stanzas" />
  </error>
</iq>
```

**Conflicting Registrations:**

```
<iq type="error">
  <query xmlns="jabber:iq:register">
    <username>Foo</username>
    <password>hunter2</password>
  </query>
  <error code="409" type="cancel">
    <conflict xmlns="urn:ietf:params:xml:ns:xmpp-stanzas" />
  </error>
</iq>
```

**Successful Registration:**

```
<iq type="result">
  <query xmlns="jabber:iq:register" />
</iq>
```

**Cases 1 and 2: Registration Requests**

Responding to registration requests depends on if the requesting user already has an account. If there is an account, the response should include the `registered` flag and the user's current registration information. Otherwise, we just send the fields for our registration form.

We will handle both cases by creating a `sendRegistrationForm` method that will create either an empty or full form depending on if we provide it with user data. Since we need to know which form fields to include (especially if we add support for the other fields specified in XEP-0077), we will also create a method `setForm` which will take the names of the fields we wish to include.

```
def plugin_init(self):
    self.description = "In-Band Registration"
    self.xep = "0077"
    self.form_fields = ('username', 'password')
```

```
... remainder of plugin_init

...

def __handleRegistration(self, iq):
    if iq['type'] == 'get':
        # Registration form requested
        userData = self.backend[iq['from']].bare
        self.sendRegistrationForm(iq, userData)

def setForm(self, *fields):
    self.form_fields = fields

def sendRegistrationForm(self, iq, userData=None):
    reg = iq['register']
    if userData is None:
        userData = {}
    else:
        reg['registered'] = True

    for field in self.form_fields:
        data = userData.get(field, '')
        if data:
            # Add field with existing data
            reg[field] = data
        else:
            # Add a blank field
            reg.addField(field)

    iq.reply().setPayload(reg.xml)
    iq.send()
```

Note how we are able to access our Registration stanza object with `iq['register']`.

## A User Backend

You might have noticed the reference to `self.backend`, which is an object that abstracts away storing and retrieving user information. Since it is not much more than a dictionary, we will leave the implementation details to the final, full source code example.

## Case 3: Unregister an Account

The next simplest case to consider is responding to a request to remove an account. If we receive a `remove` flag, we instruct the backend to remove the user's account. Since your application may need to know about when users are registered or unregistered, we trigger an event using `self.xmpp.event('unregister_user', iq)`. See the component examples below for how to respond to that event.

```
def __handleRegistration(self, iq):
    if iq['type'] == 'get':
        # Registration form requested
        userData = self.backend[iq['from']].bare
        self.sendRegistrationForm(iq, userData)
    elif iq['type'] == 'set':
        # Remove an account
        if iq['register']['remove']:
            self.backend.unregister(iq['from'].bare)
```

```
self.xmpp.event('unregistered_user', iq)
iq.reply().send()
return
```

### Case 4: Incomplete Registration

For the next case we need to check the user's registration to ensure it has all of the fields we wanted. The simple option that we will use is to loop over the field names and check each one; however, this means that all fields we send to the user are required. Adding optional fields is left to the reader.

Since we have received an incomplete form, we need to send an error message back to the user. We have to send a few different types of errors, so we will also create a `_sendError` method that will add the appropriate `error` element to the IQ reply.

```
def __handleRegistration(self, iq):
    if iq['type'] == 'get':
        # Registration form requested
        userData = self.backend[iq['from']].bare
        self.sendRegistrationForm(iq, userData)
    elif iq['type'] == 'set':
        if iq['register']['remove']:
            # Remove an account
            self.backend.unregister(iq['from'].bare)
            self.xmpp.event('unregistered_user', iq)
            iq.reply().send()
            return

        for field in self.form_fields:
            if not iq['register'][field]:
                # Incomplete Registration
                self._sendError(iq, '406', 'modify', 'not-acceptable',
                               "Please fill in all fields.")
                return

    ...

def _sendError(self, iq, code, error_type, name, text=''):
    iq.reply().setPayload(iq['register'].xml)
    iq.error()
    iq['error']['code'] = code
    iq['error']['type'] = error_type
    iq['error']['condition'] = name
    iq['error']['text'] = text
    iq.send()
```

### Cases 5 and 6: Conflicting and Successful Registration

We are down to the final decision on if we have a successful registration. We send the user's data to the backend with the `self.backend.register` method. If it returns `True`, then registration has been successful. Otherwise, there has been a conflict with usernames and registration has failed. Like with unregistering an account, we trigger an event indicating that a user has been registered by using `self.xmpp.event('registered_user', iq)`. See the component examples below for how to respond to this event.

```
def __handleRegistration(self, iq):
    if iq['type'] == 'get':
        # Registration form requested
```

```
userData = self.backend[iq['from'].bare]
self.sendRegistrationForm(iq, userData)
elif iq['type'] == 'set':
    if iq['register']['remove']:
        # Remove an account
        self.backend.unregister(iq['from'].bare)
        self.xmpp.event('unregistered_user', iq)
        iq.reply().send()
        return

    for field in self.form_fields:
        if not iq['register'][field]:
            # Incomplete Registration
            self._sendError(iq, '406', 'modify', 'not-acceptable',
                           "Please fill in all fields.")
            return

    if self.backend.register(iq['from'].bare, iq['register']):
        # Successful registration
        self.xmpp.event('registered_user', iq)
        iq.reply().setPayload(iq['register'].xml)
        iq.send()
    else:
        # Conflicting registration
        self._sendError(iq, '409', 'cancel', 'conflict',
                       "That username is already taken.")
```

### 3.4.6 Example Component Using the XEP-0077 Plugin

Alright, the moment we've been working towards - actually using our plugin to simplify our other applications. Here is a basic component that simply manages user registrations and sends the user a welcoming message when they register, and a farewell message when they delete their account.

Note that we have to register the 'xep\_0030' plugin first, and that we specified the form fields we wish to use with `self.xmpp.plugin['xep_0077'].setForm('username', 'password')`.

```
import sleekxmpp.componentxmpp

class Example(sleekxmpp.componentxmpp.ComponentXMPP):

    def __init__(self, jid, password):
        sleekxmpp.componentxmpp.ComponentXMPP.__init__(self, jid, password, 'localhost', 8888)

        self.registerPlugin('xep_0030')
        self.registerPlugin('xep_0077')
        self.plugin['xep_0077'].setForm('username', 'password')

        self.add_event_handler("registered_user", self.reg)
        self.add_event_handler("unregistered_user", self.unreg)

    def reg(self, iq):
        msg = "Welcome! %s" % iq['register']['username']
        self.sendMessage(iq['from'], msg, mfrom=self.fulljid)

    def unreg(self, iq):
        msg = "Bye! %s" % iq['register']['username']
        self.sendMessage(iq['from'], msg, mfrom=self.fulljid)
```



**Congratulations!** We now have a basic, functioning implementation of XEP-0077.

### 3.4.7 Complete Source Code for XEP-0077 Plugin

Here is a copy of a more complete implementation of the plugin we created, but with some additional registration fields implemented.

```
"""
Creating a SleekXMPP Plugin

This is a minimal implementation of XEP-0077 to serve
as a tutorial for creating SleekXMPP plugins.
"""

from sleekxmpp.plugins.base import base_plugin
from sleekxmpp.xmlstream.handler.callback import Callback
from sleekxmpp.xmlstream.matcher.xpath import MatchXPath
from sleekxmpp.xmlstream import ElementBase, ET, JID, register_stanza_plugin
from sleekxmpp import Iq
import copy

class Registration(ElementBase):
    namespace = 'jabber:iq:register'
    name = 'query'
    plugin_attrib = 'register'
    interfaces = set(('username', 'password', 'email', 'nick', 'name',
                      'first', 'last', 'address', 'city', 'state', 'zip',
                      'phone', 'url', 'date', 'misc', 'text', 'key',
                      'registered', 'remove', 'instructions'))
    sub_interfaces = interfaces

    def getRegistered(self):
        present = self.xml.find('{%s}registered' % self.namespace)
        return present is not None

    def getRemove(self):
        present = self.xml.find('{%s}remove' % self.namespace)
        return present is not None

    def setRegistered(self, registered):
        if registered:
            self.addField('registered')
        else:
            del self['registered']

    def setRemove(self, remove):
        if remove:
            self.addField('remove')
        else:
            del self['remove']

    def addField(self, name):
        itemXML = ET.Element('{%s}%s' % (self.namespace, name))
        self.xml.append(itemXML)

class UserStore(object):
```

```
def __init__(self):
    self.users = {}

def __getitem__(self, jid):
    return self.users.get(jid, None)

def register(self, jid, registration):
    username = registration['username']

    def filter_usernames(user):
        return user != jid and self.users[user]['username'] == username

    conflicts = filter(filter_usernames, self.users.keys())
    if conflicts:
        return False

    self.users[jid] = registration
    return True

def unregister(self, jid):
    del self.users[jid]

class xep_0077(base_plugin):
    """
    XEP-0077 In-Band Registration
    """

    def plugin_init(self):
        self.description = "In-Band Registration"
        self.xep = "0077"
        self.form_fields = ('username', 'password')
        self.form_instructions = ""
        self.backend = UserStore()

        self.xmpp.register_handler(
            Callback('In-Band Registration',
                    MatchXPath('{%s}iq/{jabber:iq:register}query' % self.xmpp.default_ns),
                    self.__handleRegistration))
        register_stanza_plugin(Iq, Registration)

    def post_init(self):
        base_plugin.post_init(self)
        self.xmpp['xep_0030'].add_feature("jabber:iq:register")

    def __handleRegistration(self, iq):
        if iq['type'] == 'get':
            # Registration form requested
            userData = self.backend[iq['from']].bare
            self.sendRegistrationForm(iq, userData)
        elif iq['type'] == 'set':
            if iq['register']['remove']:
                # Remove an account
                self.backend.unregister(iq['from'].bare)
                self.xmpp.event('unregistered_user', iq)
                iq.reply().send()
                return

            for field in self.form_fields:
```

```

        if not iq['register'][field]:
            # Incomplete Registration
            self._sendError(iq, '406', 'modify', 'not-acceptable',
                            "Please fill in all fields.")
            return

    if self.backend.register(iq['from'].bare, iq['register']):
        # Successful registration
        self.xmpp.event('registered_user', iq)
        iq.reply().setPayload(iq['register'].xml)
        iq.send()
    else:
        # Conflicting registration
        self._sendError(iq, '409', 'cancel', 'conflict',
                        "That username is already taken.")

def setForm(self, *fields):
    self.form_fields = fields

def setInstructions(self, instructions):
    self.form_instructions = instructions

def sendRegistrationForm(self, iq, userData=None):
    reg = iq['register']
    if userData is None:
        userData = {}
    else:
        reg['registered'] = True

    if self.form_instructions:
        reg['instructions'] = self.form_instructions

    for field in self.form_fields:
        data = userData.get(field, '')
        if data:
            # Add field with existing data
            reg[field] = data
        else:
            # Add a blank field
            reg.addField(field)

    iq.reply().setPayload(reg.xml)
    iq.send()

def _sendError(self, iq, code, error_type, name, text=''):
    iq.reply().setPayload(iq['register'].xml)
    iq.error()
    iq['error']['code'] = code
    iq['error']['type'] = error_type
    iq['error']['condition'] = name
    iq['error']['text'] = text
    iq.send()

```

## 3.5 How to Use Stream Features

## 3.6 How SASL Authentication Works

## 3.7 Using Stream Handlers and Matchers

## 3.8 Plugin Guides

### 3.8.1 XEP-0030: Working with Service Discovery

XMPP networks can be composed of many individual clients, components, and servers. Determining the JIDs for these entities and the various features they may support is the role of [XEP-0030](#), [Service Discovery](#), or “disco” for short.

Every XMPP entity may possess what are called nodes. A node is just a name for some aspect of an XMPP entity. For example, if an XMPP entity provides [Ad-Hoc Commands](#), then it will have a node named `http://jabber.org/protocol/commands` which will contain information about the commands provided. Other agents using these ad-hoc commands will interact with the information provided by this node. Note that the node name is just an identifier; there is no inherent meaning.

Working with service discovery is about creating and querying these nodes. According to XEP-0030, a node may contain three types of information: identities, features, and items. (Further, extensible, information types are defined in [XEP-0128](#), but they are not yet implemented by SleekXMPP.) SleekXMPP provides methods to configure each of these node attributes.

#### Configuring Service Discovery

The design focus for the XEP-0030 plug-in is handling info and items requests in a dynamic fashion, allowing for complex policy decisions of who may receive information and how much, or use alternate backend storage mechanisms for all of the disco data. To do this, each action that the XEP-0030 plug-in performs is handed off to what is called a “node handler,” which is just a callback function. These handlers are arranged in a hierarchy that allows for a single handler to manage an entire domain of JIDs (say for a component), while allowing other handler functions to override that global behaviour for certain JIDs, or even further limited to only certain JID and node combinations.

#### The Dynamic Handler Hierarchy

- `global`: (JID is None, node is None)

Handlers assigned at this level for an action (such as `add_feature`) provide a global default behaviour when the action is performed.

- `jid`: (JID assigned, node is None)

At this level, handlers provide a default behaviour for actions affecting any node owned by the JID in question. This level is most useful for component connections; there is effectively no difference between this and the global level when using a client connection.

- `node`: (JID assigned, node assigned)

A handler for this level is responsible for carrying out an action for only one node, and is the most specific handler type available. These types of handlers will be most useful for “special” nodes that require special

processing different than others provided by the JID, such as using access control lists, or consolidating data from other nodes.

## Default Static Handlers

The XEP-0030 plug-in provides a default set of handlers that work using in-memory disco stanzas. Each handler simply performs the appropriate lookup or storage operation using these stanzas without doing any complex operations such as checking an ACL, etc.

You may find it necessary at some point to revert a particular node or JID to using the default, static handlers. To do so, use the method `make_static()`. You may also elect to only convert a given set of actions instead.

## Creating a Node Handler

Every node handler receives three arguments: the JID, the node, and a data parameter that will contain the relevant information for carrying out the handler's action, typically a dictionary.

The JID will always have a value, defaulting to `xmpp.boundjid.full` for components or `xmpp.boundjid.bare` for clients. The node value may be `None` or a string.

Only handlers for the actions `get_info` and `get_items` need to have return values. For these actions, `DiscoInfo` or `DiscoItems` stanzas are expected as output. It is also acceptable for handlers for these actions to generate an `XMPPError` exception when necessary.

**Example Node Handler:** Here is one of the built-in default handlers as an example:

```
def add_identity(self, jid, node, data):
    """
    Add a new identity to the JID/node combination.

    The data parameter may provide:
        category -- The general category to which the agent belongs.
        itype    -- A more specific designation with the category.
        name     -- Optional human readable name for this identity.
        lang     -- Optional standard xml:lang value.
    """
    self.add_node(jid, node)
    self.nodes[(jid, node)][ 'info' ].add_identity(
        data.get('category', ''),
        data.get('itype', ''),
        data.get('name', None),
        data.get('lang', None))
```

## Adding Identities, Features, and Items

In order to maintain some backwards compatibility, the methods `add_identity`, `add_feature`, and `add_item` do not follow the method signature pattern of the other API methods (i.e. `jid`, `node`, then other options), but rather retain the parameter orders from previous plug-in versions.

## Adding an Identity

Adding an identity may be done using either the older positional notation, or with keyword parameters. The example below uses the keyword arguments, but in the same order as expected using positional arguments.

```
xmpp['xep_0030'].add_identity(category='client',
                               itype='bot',
                               name='Sleek',
                               node='foo',
                               jid=xmpp.boundjid.full,
                               lang='no')
```

The JID and node values determine which handler will be used to perform the `add_identity` action.

The `lang` parameter allows for adding localized versions of identities using the `xml:lang` attribute.

### Adding a Feature

The position ordering for `add_feature()` is to include the feature, then specify the node and then the JID. The JID and node values determine which handler will be used to perform the `add_feature` action.

```
xmpp['xep_0030'].add_feature(feature='jabber:x:data',
                              node='foo',
                              jid=xmpp.boundjid.full)
```

### Adding an Item

The parameters to `add_item()` are potentially confusing due to the fact that adding an item requires two JID and node combinations: the JID and node of the item itself, and the JID and node that will own the item.

```
xmpp['xep_0030'].add_item(jid='myitemjid@example.com',
                           name='An Item!',
                           node='owner_node',
                           subnode='item_node',
                           ijid=xmpp.boundjid.full)
```

---

**Note:** In this case, the owning JID and node are provided with the parameters `ijid` and `node`.

---

### Performing Disco Queries

The methods `get_info()` and `get_items()` are used to query remote JIDs and their nodes for disco information. Since these methods are wrappers for sending `Iq` stanzas, they also accept all of the parameters of the `Iq.send()` method. The `get_items()` method may also accept the boolean parameter `iterator`, which when set to `True` will return an iterator object using the [XEP-0059](#) plug-in.

```
info = self['xep_0030'].get_info(jid='foo@example.com',
                                 node='bar',
                                 ifrom='baz@mycomponent.example.com',
                                 block=True,
                                 timeout=30)

items = self['xep_0030'].get_info(jid='foo@example.com',
                                  node='bar',
                                  iterator=True)
```

For more examples on how to use basic disco queries, check the `disco_browser.py` example in the examples directory.

## Local Queries

In some cases, it may be necessary to query the contents of a node owned by the client itself, or one of a component's many JIDs. The same method is used as for normal queries, with two differences. First, the parameter `local=True` must be used. Second, the return value will be a `DiscoInfo` or `DiscoItems` stanza, not a full `Iq` stanza.

```
info = self['xep_0030'].get_info(node='foo', local=True)
items = self['xep_0030'].get_items(jid='somejid@mycomponent.example.com',
                                   node='bar',
                                   local=True)
```

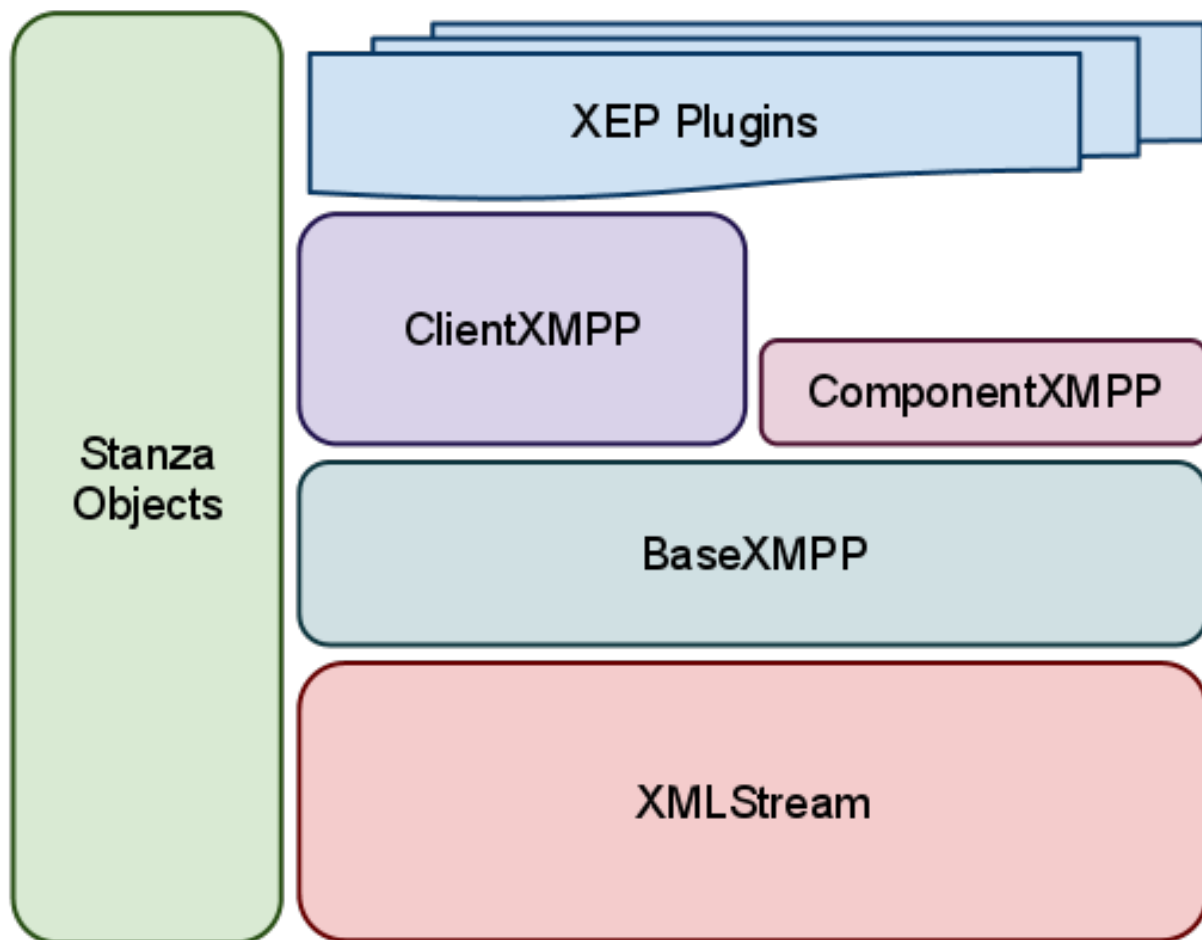




# SLEEKXMPP ARCHITECTURE AND DESIGN

## 4.1 SleekXMPP Architecture

The core of SleekXMPP is contained in four classes: `XMLStream`, `BaseXMPP`, `ClientXMPP`, and `ComponentXMPP`. Along side this stack is a library for working with XML objects that eliminates most of the tedium of creating/manipulating XML.



### 4.1.1 The Foundation: XMLStream

`XMLStream` is a mostly XMPP-agnostic class whose purpose is to read and write from a bi-directional XML stream. It also allows for callback functions to execute when XML matching given patterns is received; these callbacks are also referred to as *stream handlers*. The class also provides a basic eventing system which can be triggered either manually or on a timed schedule.

#### The Main Threads

`XMLStream` instances run using at least three background threads: the send thread, the read thread, and the scheduler thread. The send thread is in charge of monitoring the send queue and writing text to the outgoing XML stream. The read thread pulls text off of the incoming XML stream and stores the results in an event queue. The scheduler thread is used to emit events after a given period of time.

Additionally, the main event processing loop may be executed in its own thread if SleekXMPP is being used in the background for another application.

Short-lived threads may also be spawned as requested for threaded *event handlers*.

#### How XML Text is Turned into Action

To demonstrate the flow of information, let's consider what happens when this bit of XML is received (with an assumed namespace of `jabber:client`):

```
<message to="user@example.com" from="friend@example.net">
  <body>Hej!</body>
</message>
```

1. **Convert XML strings into objects.**

Incoming text is parsed and converted into XML objects (using `ElementTree`) which are then wrapped into what are referred to as *Stanza objects*. The appropriate class for the new object is determined using a map of namespaced element names to classes.

Our incoming XML is thus turned into a `Message` *stanza object* because the namespaced element name `{jabber:client}message` is associated with the class `Message`.

2. **Match stanza objects to callbacks.**

These objects are then compared against the stored patterns associated with the registered callback handlers. For each match, a copy of the *stanza object* is paired with a reference to the handler and placed into the event queue.

Our `Message` object is thus paired with the message stanza handler `BaseXMPP._handle_message()` to create the tuple:

```
('stanza', stanza_obj, handler)
```

3. **Process the event queue.**

The event queue is the heart of SleekXMPP. Nearly every action that takes place is first inserted into this queue, whether that be received stanzas, custom events, or scheduled events.

When the stanza is pulled out of the event queue with an associated callback, the callback function is executed with the stanza as its only parameter.

**Warning:** The callback, aka *stream handler*, is executed in the main event processing thread. If the handler blocks, event processing will also block.

#### 4. Raise Custom Events

Since a *stream handler* shouldn't block, if extensive processing for a stanza is required (such as needing to send and receive an `Iq` stanza), then custom events must be used. These events are not explicitly tied to the incoming XML stream and may be raised at any time. Importantly, these events may be handled in their own thread.

When the event is raised, a copy of the stanza is created for each handler registered for the event. In contrast to *stream handlers*, these functions are referred to as *event handlers*. Each stanza/handler pair is then put into the event queue.

---

**Note:** It is possible to skip the event queue and process an event immediately by using `direct=True` when raising the event.

---

The code for `BaseXMPP._handle_message()` follows this pattern, and raises a 'message' event:

```
self.event('message', msg)
```

The event call then places the message object back into the event queue paired with an *event handler*:

```
('event', 'message', msg_copy1, custom_event_handler_1)
('event', 'message', msg_copy2, custom_event_handler_2)
```

#### 5. Process Custom Events

The stanza and *event handler* are then pulled from the event queue, and the handler is executed, passing the stanza as its only argument. If the handler was registered as threaded, then a new thread will be spawned for it.

---

**Note:** Events may be raised without needing *stanza objects*. For example, you could use `self.event('custom', {'a': 'b'})`. You don't even need any arguments: `self.event('no_parameters')`. However, every event handler **MUST** accept at least one argument.

---

Finally, after a long trek, our message is handed off to the user's custom handler in order to do awesome stuff:

```
msg.reply()
msg['body'] = "Hey! This is awesome!"
msg.send()
```

### 4.1.2 Raising XMPP Awareness: BaseXMPP

While `XMLStream` attempts to shy away from anything too XMPP specific, `BaseXMPP`'s sole purpose is to provide foundational support for sending and receiving XMPP stanzas. This support includes registering the basic message, presence, and iq stanzas, methods for creating and sending stanzas, and default handlers for incoming messages and keeping track of presence notifications.

The plugin system for adding new XEP support is also maintained by `BaseXMPP`.

### 4.1.3 ClientXMPP

`ClientXMPP` extends `BaseXMPP` with additional logic for connecting to an XMPP server by performing DNS lookups. It also adds support for stream features such as STARTTLS and SASL.

#### 4.1.4 ComponentXMPP

`ComponentXMPP` is only a thin layer on top of `BaseXMPP` that implements the component handshake protocol.

### 4.2 Plugin Architecture

---

# API REFERENCE

## 5.1 Event Index

### changed\_status

- **Data:** Presence
- **Source:** RosterItem

Triggered when a presence stanza is received from a JID with a show type different than the last presence stanza from the same JID.

### changed\_subscription

- **Data:** Presence
- **Source:** BaseXMPP

Triggered whenever a presence stanza with a type of subscribe, subscribed, unsubscribe, or unsubscribed is received.

Note that if the values `xmpp.auto_authorize` and `xmpp.auto_subscribe` are set to `True` or `False`, and not `None`, then SleekXMPP will either accept or reject all subscription requests before your event handlers are called. Set these values to `None` if you wish to make more complex subscription decisions.

### chatstate\_active

- **Data:**
- **Source:**

### chatstate\_composing

- **Data:**
- **Source:**

### chatstate\_gone

- **Data:**
- **Source:**

### chatstate\_inactive

- **Data:**
- **Source:**

### chatstate\_paused

- **Data:**
- **Source:**

**connected**

- **Data:** {}
- **Source:** XMLstream

Signal that a connection has been made with the XMPP server, but a session has not yet been established.

**connection\_failed**

- **Data:** {} or Failure Stanza if available
- **Source:** XMLstream

Signal that a connection can not be established after number of attempts.

**disco\_info**

- **Data:** DiscoInfo
- **Source:** xep\_0030

Triggered whenever a disco#info result stanza is received.

**disco\_items**

- **Data:** DiscoItems
- **Source:** xep\_0030

Triggered whenever a disco#items result stanza is received.

**disconnected**

- **Data:** {}
- **Source:** XMLstream

Signal that the connection with the XMPP server has been lost.

**entity\_time**

- **Data:**
- **Source:**

**failed\_auth**

- **Data:** {}
- **Source:** ClientXMPP, xep\_0078

Signal that the server has rejected the provided login credentials.

**gmail\_messages**

- **Data:** Iq
- **Source:** gmail\_notify

Signal that there are unread emails for the Gmail account associated with the current XMPP account.

**gmail\_notify**

- **Data:** {}
- **Source:** gmail\_notify

Signal that there are unread emails for the Gmail account associated with the current XMPP account.

**got\_offline**

- **Data:** Presence
- **Source:** RosterItem

Signal that an unavailable presence stanza has been received from a JID.

**got\_online**

- **Data:** Presence
- **Source:** RosterItem

If a presence stanza is received from a JID which was previously marked as offline, and the presence has a show type of 'chat', 'dnd', 'away', or 'xa', then this event is triggered as well.

**groupchat\_direct\_invite**

- **Data:** Message
- **Source:** direct

**groupchat\_invite**

- **Data:**
- **Source:**

**groupchat\_message**

- **Data:** Message
- **Source:** xep\_0045

Triggered whenever a message is received from a multi-user chat room.

**groupchat\_presence**

- **Data:** Presence
- **Source:** xep\_0045

Triggered whenever a presence stanza is received from a user in a multi-user chat room.

**groupchat\_subject**

- **Data:** Message
- **Source:** xep\_0045

Triggered whenever the subject of a multi-user chat room is changed, or announced when joining a room.

**killed**

- **Data:**
- **Source:**

**last\_activity**

- **Data:**
- **Source:**

**message**

- **Data:** Message

- **Source:** BaseXMPP

Makes the contents of message stanzas available whenever one is received. Be sure to check the message type in order to handle error messages.

### message\_form

- **Data:** Form
- **Source:** xep\_0004

Currently the same as *message\_xform*.

### message\_xform

- **Data:** Form
- **Source:** xep\_0004

Triggered whenever a data form is received inside a message.

### muc::[room]::got\_offline

- **Data:**
- **Source:**

### muc::[room]::got\_online

- **Data:**
- **Source:**

### muc::[room]::message

- **Data:**
- **Source:**

### muc::[room]::presence

- **Data:**
- **Source:**

### presence\_available

- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘available’ is received.

### presence\_error

- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘error’ is received.

### presence\_form

- **Data:** Form
- **Source:** xep\_0004

This event is present in the XEP-0004 plugin code, but is currently not used.

### presence\_probe



- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘probe’ is received.

#### **presence\_subscribe**

- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘subscribe’ is received.

#### **presence\_subscribed**

- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘subscribed’ is received.

#### **presence\_unavailable**

- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘unavailable’ is received.

#### **presence\_unsubscribe**

- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘unsubscribe’ is received.

#### **presence\_unsubscribed**

- **Data:** Presence
- **Source:** BaseXMPP

A presence stanza with a type of ‘unsubscribed’ is received.

#### **roster\_update**

- **Data:** Roster
- **Source:** ClientXMPP

An IQ result containing roster entries is received.

#### **sent\_presence**

- **Data:** {}
- **Source:** Roster

Signal that an initial presence stanza has been written to the XML stream.

#### **session\_end**

- **Data:** {}
- **Source:** XMLstream

Signal that a connection to the XMPP server has been lost and the current stream session has ended. Currently equivalent to *disconnected*, but future implementation of XEP-0198: *Stream Management* will distinguish the two events.

Plugins that maintain session-based state should clear themselves when this event is fired.

#### **session\_start**

- **Data:** {}
- **Source:** ClientXMPP, ComponentXMPP XEP-0078

Signal that a connection to the XMPP server has been made and a session has been established.

#### **socket\_error**

- **Data:** Socket exception object
- **Source:** XMLstream

#### **stream\_error**

- **Data:** StreamError
- **Source:** BaseXMPP

## 5.2 ClientXMPP

**class** sleekxmpp.clientxmpp.**ClientXMPP** (*jid, password, plugin\_config={}, plugin\_whitelist=[], escape\_quotes=True, sasl\_mech=None, lang=u'en'*)

SleekXMPP's client class. (Use only for good, not for evil.)

Typical use pattern:

```
xmpp = ClientXMPP('user@server.tld/resource', 'password')
# ... Register plugins and event handlers ...
xmpp.connect()
xmpp.process(block=False) # block=True will block the current
                           # thread. By default, block=False
```

#### **Parameters**

- **jid** – The JID of the XMPP user account.
- **password** – The password for the XMPP user account.
- **ssl** – **Deprecated.**
- **plugin\_config** – A dictionary of plugin configurations.
- **plugin\_whitelist** – A list of approved plugins that will be loaded when calling `register_plugins()`.
- **escape\_quotes** – **Deprecated.**

**connect** (*address=(), reattempt=True, use\_tls=True, use\_ssl=False*)

Connect to the XMPP server.

When no address is given, a SRV lookup for the server will be attempted. If that fails, the server user in the JID will be used.

:param address – A tuple containing the server's host and port. :param reattempt: If `True`, repeat attempting to connect if an

error occurs. Defaults to `True`.

#### Parameters

- **use\_tls** – Indicates if TLS should be used for the connection. Defaults to `True`.
- **use\_ssl** – Indicates if the older SSL connection method should be used. Defaults to `False`.

#### **delRosterItem** (*jid*)

Remove an item from the roster.

This is done by setting its subscription status to `'remove'`.

**Parameters** **jid** – The JID of the item to remove.

#### **del\_roster\_item** (*jid*)

Remove an item from the roster.

This is done by setting its subscription status to `'remove'`.

**Parameters** **jid** – The JID of the item to remove.

#### **getRoster** (*block=True, timeout=None, callback=None*)

Request the roster from the server.

#### Parameters

- **block** – Specify if the roster request will block until a response is received, or a timeout occurs. Defaults to `True`.
- **timeout** –  
**The length of time (in seconds) to wait for a response** before continuing if blocking is used. Defaults to `response_timeout`.
- **callback** – Optional reference to a stream handler function. Will be executed when the roster is received. Implies `block=False`.

#### **get\_roster** (*block=True, timeout=None, callback=None*)

Request the roster from the server.

#### Parameters

- **block** – Specify if the roster request will block until a response is received, or a timeout occurs. Defaults to `True`.
- **timeout** –  
**The length of time (in seconds) to wait for a response** before continuing if blocking is used. Defaults to `response_timeout`.
- **callback** – Optional reference to a stream handler function. Will be executed when the roster is received. Implies `block=False`.

#### **registerFeature** (*name, handler, restart=False, order=5000*)

Register a stream feature handler.

#### Parameters

- **name** – The name of the stream feature.

- **handler** – The function to execute if the feature is received.
- **restart** – Indicates if feature processing should halt with this feature. Defaults to `False`.
- **order** – The relative ordering in which the feature should be negotiated. Lower values will be attempted earlier when available.

**register\_feature** (*name, handler, restart=False, order=5000*)

Register a stream feature handler.

#### Parameters

- **name** – The name of the stream feature.
- **handler** – The function to execute if the feature is received.
- **restart** – Indicates if feature processing should halt with this feature. Defaults to `False`.
- **order** – The relative ordering in which the feature should be negotiated. Lower values will be attempted earlier when available.

**updateRoster** (*jid, \*\*kwargs*)

Add or change a roster item.

#### Parameters

- **jid** – The JID of the entry to modify.
- **name** – The user's nickname for this JID.
- **subscription** – The subscription status. May be one of `'to'`, `'from'`, `'both'`, or `'none'`. If set to `'remove'`, the entry will be deleted.
- **groups** – The roster groups that contain this item.
- **block** – Specify if the roster request will block until a response is received, or a timeout occurs. Defaults to `True`.
- **timeout** –  
**The length of time (in seconds) to wait** for a response before continuing if blocking is used. Defaults to `response_timeout`.
- **callback** – Optional reference to a stream handler function. Will be executed when the roster is received. Implies `block=False`.

**update\_roster** (*jid, \*\*kwargs*)

Add or change a roster item.

#### Parameters

- **jid** – The JID of the entry to modify.
- **name** – The user's nickname for this JID.
- **subscription** – The subscription status. May be one of `'to'`, `'from'`, `'both'`, or `'none'`. If set to `'remove'`, the entry will be deleted.
- **groups** – The roster groups that contain this item.
- **block** – Specify if the roster request will block until a response is received, or a timeout occurs. Defaults to `True`.
- **timeout** –

The length of time (in seconds) to wait for a response before continuing if blocking is used. Defaults to

`response_timeout`.

- **callback** – Optional reference to a stream handler function. Will be executed when the roster is received. Implies `block=False`.

## 5.3 ComponentXMPP

```
class sleekxmpp.componentxmpp.ComponentXMPP(jid, secret, host=None, port=None, plugin_config={}, plugin_whitelist=[], use_jc_ns=False)
```

SleekXMPP's basic XMPP server component.

Use only for good, not for evil.

### Parameters

- **jid** – The JID of the component.
- **secret** – The secret or password for the component.
- **host** – The server accepting the component.
- **port** – The port used to connect to the server.
- **plugin\_config** – A dictionary of plugin configurations.
- **plugin\_whitelist** – A list of approved plugins that will be loaded when calling `register_plugins()`.
- **use\_jc\_ns** – Indicates if the 'jabber:client' namespace should be used instead of the standard 'jabber:component:accept' namespace. Defaults to `False`.

```
connect(host=None, port=None, use_ssl=False, use_tls=False, reattempt=True)
```

Connect to the server.

Setting `reattempt` to `True` will cause connection attempts to be made every second until a successful connection is established.

### Parameters

- **host** – The name of the desired server for the connection. Defaults to `server_host`.
- **port** – Port to connect to on the server. Defaults to `server_port`.
- **use\_ssl** – Flag indicating if SSL should be used by connecting directly to a port using SSL.
- **use\_tls** – Flag indicating if TLS should be used, allowing for connecting to a port without using SSL immediately and later upgrading the connection.
- **reattempt** – Flag indicating if the socket should reconnect after disconnections.

```
incoming_filter(xml)
```

Pre-process incoming XML stanzas by converting any 'jabber:client' namespaced elements to the component's default namespace.

**Parameters** `xml` – The XML stanza to pre-process.

```
start_stream_handler(xml)
```

Once the streams are established, attempt to handshake with the server to be accepted as a component.

**Parameters** `xml` – The incoming stream's root element.

## 5.4 BaseXMPP

**class** sleekxmpp.basexmpp.**BaseXMPP** (*jid=u'', default\_ns=u'jabber:client'*)

The BaseXMPP class adapts the generic XMLStream class for use with XMPP. It also provides a plugin mechanism to easily extend and add support for new XMPP features.

**Parameters** **default\_ns** – Ensure that the correct default XML namespace is used during initialization.

**Iq** (*\*args, \*\*kwargs*)

Create an Iq stanza associated with this stream.

**Message** (*\*args, \*\*kwargs*)

Create a Message stanza associated with this stream.

**Presence** (*\*args, \*\*kwargs*)

Create a Presence stanza associated with this stream.

**api = None**

The API registry is a way to process callbacks based on JID+node combinations. Each callback in the registry is marked with:

- An API name, e.g. xep\_0030
- The name of an action, e.g. get\_info
- The JID that will be affected
- The node that will be affected

API handlers with no JID or node will act as global handlers, while those with a JID and no node will service all nodes for a JID, and handlers with both a JID and node will be used only for that specific combination. The handler that provides the most specificity will be used.

**auto\_authorize**

Auto accept or deny subscription requests.

If **True**, auto accept subscription requests. If **False**, auto deny subscription requests. If **None**, don't automatically respond.

**auto\_subscribe**

Auto send requests for mutual subscriptions.

If **True**, auto send mutual subscription requests.

**boundjid = None**

The JabberID (JID) used by this connection, as set after session binding. This may even be a different bare JID than what was requested.

**client\_roster = None**

The single roster for the bound JID. This is the equivalent of:

```
self.roster[self.boundjid.bare]
```

**exception** (*exception*)

Process any uncaught exceptions, notably `IqError` and `IqTimeout` exceptions.

**Parameters** **exception** – An unhandled `Exception` object.

**fulljid**

Attribute accessor for full jid

**get** (*key*, *default*)

Return a plugin given its name, if it has been registered.

**is\_component** = None

The distinction between clients and components can be important, primarily for choosing how to handle the 'to' and 'from' JIDs of stanzas.

**jid**

Attribute accessor for bare jid

**makeIq** (*id=0*, *ifrom=None*, *ito=None*, *itype=None*, *iquery=None*)

Create a new Iq stanza with a given Id and from JID.

#### Parameters

- **id** – An ideally unique ID value for this stanza thread. Defaults to 0.
- **ifrom** – The from JID to use for this stanza.
- **ito** – The destination JID for this stanza.
- **itype** – The Iq's type, one of: 'get', 'set', 'result', or 'error'.
- **iquery** – Optional namespace for adding a query element.

**makeIqError** (*id*, *type=u'cancel'*, *condition=u'feature-not-implemented'*, *text=None*, *ito=None*, *ifrom=None*, *iq=None*)

Create an Iq stanza of type 'error'.

#### Parameters

- **id** – An ideally unique ID value. May use `new_id()`.
- **type** – The type of the error, such as 'cancel' or 'modify'. Defaults to 'cancel'.
- **condition** – The error condition. Defaults to 'feature-not-implemented'.
- **text** – A message describing the cause of the error.
- **ito** – The destination JID for this stanza.
- **ifrom** – The 'from' JID to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**makeIqGet** (*queryxmlns=None*, *ito=None*, *ifrom=None*, *iq=None*)

Create an Iq stanza of type 'get'.

Optionally, a query element may be added.

#### Parameters

- **queryxmlns** – The namespace of the query to use.
- **ito** – The destination JID for this stanza.
- **ifrom** – The 'from' JID to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**makeIqQuery** (*iq=None*, *xmlns=u''*, *ito=None*, *ifrom=None*)

Create or modify an Iq stanza to use the given query namespace.

#### Parameters

- **iq** – Optionally use an existing stanza instead of generating a new one.
- **xmlns** – The query's namespace.

- **ito** – The destination [JID](#) for this stanza.
- **ifrom** – The 'from' [JID](#) to use for this stanza.

**makeIqResult** (*id=None, ito=None, ifrom=None, iq=None*)

Create an Iq stanza of type 'result' with the given ID value.

#### Parameters

- **id** – An ideally unique ID value. May use `new_id()`.
- **ito** – The destination [JID](#) for this stanza.
- **ifrom** – The 'from' [JID](#) to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**makeIqSet** (*sub=None, ito=None, ifrom=None, iq=None*)

Create an Iq stanza of type 'set'.

Optionally, a substanza may be given to use as the stanza's payload.

#### Parameters

- **sub** – Either an [ElementBase](#) stanza object or an [Element](#) XML object to use as the Iq's payload.
- **ito** – The destination [JID](#) for this stanza.
- **ifrom** – The 'from' [JID](#) to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**makeMessage** (*mto, mbody=None, msubject=None, mtype=None, mhtml=None, mfrom=None, mnick=None*)

Create and initialize a new Message stanza.

#### Parameters

- **mto** – The recipient of the message.
- **mbody** – The main contents of the message.
- **msubject** – Optional subject for the message.
- **mtype** – The message's type, such as 'chat' or 'groupchat'.
- **mhtml** – Optional HTML body content in the form of a string.
- **mfrom** – The sender of the message. if sending from a client, be aware that some servers require that the full JID of the sender be used.
- **mnick** – Optional nickname of the sender.

**makePresence** (*pshow=None, pstatus=None, ppriority=None, pto=None, ptype=None, pfrom=None, pnick=None*)

Create and initialize a new Presence stanza.

#### Parameters

- **pshow** – The presence's show value.
- **pstatus** – The presence's status message.
- **ppriority** – This connection's priority.
- **pto** – The recipient of a directed presence.
- **ptype** – The type of presence, such as 'subscribe'.



- **pfrom** – The sender of the presence.
- **pnick** – Optional nickname of the presence's sender.

**makeQueryRoster** (*iq=None*)

Create a roster query element.

**Parameters** **iq** – Optionally use an existing stanza instead of generating a new one.

**make\_iq** (*id=0, ifrom=None, ito=None, itype=None, iquery=None*)

Create a new Iq stanza with a given Id and from JID.

**Parameters**

- **id** – An ideally unique ID value for this stanza thread. Defaults to 0.
- **ifrom** – The from JID to use for this stanza.
- **ito** – The destination JID for this stanza.
- **itype** – The Iq's type, one of: 'get', 'set', 'result', or 'error'.
- **iquery** – Optional namespace for adding a query element.

**make\_iq\_error** (*id, type=u'cancel', condition=u'feature-not-implemented', text=None, ito=None, ifrom=None, iq=None*)

Create an Iq stanza of type 'error'.

**Parameters**

- **id** – An ideally unique ID value. May use `new_id()`.
- **type** – The type of the error, such as 'cancel' or 'modify'. Defaults to 'cancel'.
- **condition** – The error condition. Defaults to 'feature-not-implemented'.
- **text** – A message describing the cause of the error.
- **ito** – The destination JID for this stanza.
- **ifrom** – The 'from' JID to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**make\_iq\_get** (*queryxmlns=None, ito=None, ifrom=None, iq=None*)

Create an Iq stanza of type 'get'.

Optionally, a query element may be added.

**Parameters**

- **queryxmlns** – The namespace of the query to use.
- **ito** – The destination JID for this stanza.
- **ifrom** – The 'from' JID to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**make\_iq\_query** (*iq=None, xmlns=u'', ito=None, ifrom=None*)

Create or modify an Iq stanza to use the given query namespace.

**Parameters**

- **iq** – Optionally use an existing stanza instead of generating a new one.
- **xmlns** – The query's namespace.
- **ito** – The destination JID for this stanza.

- **ifrom** – The 'from' JID to use for this stanza.

**make\_iq\_result** (*id=None, ito=None, ifrom=None, iq=None*)

Create an Iq stanza of type 'result' with the given ID value.

#### Parameters

- **id** – An ideally unique ID value. May use `new_id()`.
- **ito** – The destination JID for this stanza.
- **ifrom** – The 'from' JID to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**make\_iq\_set** (*sub=None, ito=None, ifrom=None, iq=None*)

Create an Iq stanza of type 'set'.

Optionally, a substanza may be given to use as the stanza's payload.

#### Parameters

- **sub** – Either an `ElementBase` stanza object or an `Element` XML object to use as the Iq's payload.
- **ito** – The destination JID for this stanza.
- **ifrom** – The 'from' JID to use for this stanza.
- **iq** – Optionally use an existing stanza instead of generating a new one.

**make\_message** (*mto, mbody=None, msubject=None, mtype=None, mhtml=None, mfrom=None, mnick=None*)

Create and initialize a new Message stanza.

#### Parameters

- **mto** – The recipient of the message.
- **mbody** – The main contents of the message.
- **msubject** – Optional subject for the message.
- **mtype** – The message's type, such as 'chat' or 'groupchat'.
- **mhtml** – Optional HTML body content in the form of a string.
- **mfrom** – The sender of the message. if sending from a client, be aware that some servers require that the full JID of the sender be used.
- **mnick** – Optional nickname of the sender.

**make\_presence** (*pshow=None, pstatus=None, ppriority=None, pto=None, ptype=None, pfrom=None, pnick=None*)

Create and initialize a new Presence stanza.

#### Parameters

- **pshow** – The presence's show value.
- **pstatus** – The presence's status message.
- **ppriority** – This connection's priority.
- **pto** – The recipient of a directed presence.
- **ptype** – The type of presence, such as 'subscribe'.
- **pfrom** – The sender of the presence.

- **pnick** – Optional nickname of the presence’s sender.

**make\_query\_roster** (*iq=None*)

Create a roster query element.

**Parameters** **iq** – Optionally use an existing stanza instead of generating a new one.

**max\_redirects = None**

The maximum number of consecutive see-other-host redirections that will be followed before quitting.

**plugin = None**

A dictionary mapping plugin names to plugins.

**plugin\_config = None**

Configuration options for whitelisted plugins. If a plugin is registered without any configuration, and there is an entry here, it will be used.

**plugin\_whitelist = None**

A list of plugins that will be loaded if `register_plugins()` is called.

**process** (*\*args, \*\*kwargs*)

Initialize plugins and begin processing the XML stream.

The number of threads used for processing stream events is determined by `HANDLER_THREADS`.

#### Parameters

- **block** (*bool*) – If `False`, then event dispatcher will run in a separate thread, allowing for the stream to be used in the background for another application. Otherwise, `process(block=True)` blocks the current thread. Defaults to `False`.
- **threaded** (*bool*) – **DEPRECATED** If `True`, then event dispatcher will run in a separate thread, allowing for the stream to be used in the background for another application. Defaults to `True`. This does **not** mean that no threads are used at all if `threaded=False`.

Regardless of these threading options, these threads will always exist:

- The event queue processor
- The send queue processor
- The scheduler

**registerPlugin** (*plugin, pconfig={}, module=None*)

Register and configure a plugin for use in this stream.

#### Parameters

- **plugin** – The name of the plugin class. Plugin names must be unique.
- **pconfig** – A dictionary of configuration data for the plugin. Defaults to an empty dictionary.
- **module** – Optional reference to the module containing the plugin class if using custom plugins.

**register\_plugin** (*plugin, pconfig={}, module=None*)

Register and configure a plugin for use in this stream.

#### Parameters

- **plugin** – The name of the plugin class. Plugin names must be unique.
- **pconfig** – A dictionary of configuration data for the plugin. Defaults to an empty dictionary.

- **module** – Optional reference to the module containing the plugin class if using custom plugins.

**register\_plugins()**

Register and initialize all built-in plugins.

Optionally, the list of plugins loaded may be limited to those contained in `plugin_whitelist`.

Plugin configurations stored in `plugin_config` will be used.

**requested\_jid = None**

The JabberID (JID) requested for this connection.

**resource**

Attribute accessor for `jid` resource

**roster = None**

The main roster object. This roster supports multiple owner JIDs, as in the case for components. For clients which only have a single JID, see `client_roster`.

**sendMessage(mto, mbody, msubject=None, mtype=None, mhtml=None, mfrom=None, mnick=None)**

Create, initialize, and send a new `Message` stanza.

**Parameters**

- **mto** – The recipient of the message.
- **mbody** – The main contents of the message.
- **msubject** – Optional subject for the message.
- **mtype** – The message's type, such as `'chat'` or `'groupchat'`.
- **mhtml** – Optional HTML body content in the form of a string.
- **mfrom** – The sender of the message. if sending from a client, be aware that some servers require that the full JID of the sender be used.
- **mnick** – Optional nickname of the sender.

**sendPresence(pshow=None, pstatus=None, ppriority=None, pto=None, pfrom=None, ptype=None, pnick=None)**

Create, initialize, and send a new `Presence` stanza.

**Parameters**

- **pshow** – The presence's show value.
- **pstatus** – The presence's status message.
- **ppriority** – This connection's priority.
- **pto** – The recipient of a directed presence.
- **ptype** – The type of presence, such as `'subscribe'`.
- **pfrom** – The sender of the presence.
- **pnick** – Optional nickname of the presence's sender.

**sendPresenceSubscription(pto, pfrom=None, ptype=u'subscribe', pnick=None)**

Create, initialize, and send a new `Presence` stanza of type `'subscribe'`.

**Parameters**

- **pto** – The recipient of a directed presence.
- **pfrom** – The sender of the presence.

- **ptype** – The type of presence, such as 'subscribe'.
- **pnick** – Optional nickname of the presence's sender.

**send\_message** (*mto*, *mbody*, *msubject=None*, *mtype=None*, *mhtml=None*, *mfrom=None*, *mnick=None*)

Create, initialize, and send a new Message stanza.

#### Parameters

- **mto** – The recipient of the message.
- **mbody** – The main contents of the message.
- **msubject** – Optional subject for the message.
- **mtype** – The message's type, such as 'chat' or 'groupchat'.
- **mhtml** – Optional HTML body content in the form of a string.
- **mfrom** – The sender of the message. if sending from a client, be aware that some servers require that the full JID of the sender be used.
- **mnick** – Optional nickname of the sender.

**send\_presence** (*pshow=None*, *pstatus=None*, *ppriority=None*, *pto=None*, *pfrom=None*, *ptype=None*, *pnick=None*)

Create, initialize, and send a new Presence stanza.

#### Parameters

- **pshow** – The presence's show value.
- **pstatus** – The presence's status message.
- **ppriority** – This connection's priority.
- **pto** – The recipient of a directed presence.
- **ptype** – The type of presence, such as 'subscribe'.
- **pfrom** – The sender of the presence.
- **pnick** – Optional nickname of the presence's sender.

**send\_presence\_subscription** (*pto*, *pfrom=None*, *ptype=u'subscribe'*, *pnick=None*)

Create, initialize, and send a new Presence stanza of type 'subscribe'.

#### Parameters

- **pto** – The recipient of a directed presence.
- **pfrom** – The sender of the presence.
- **ptype** – The type of presence, such as 'subscribe'.
- **pnick** – Optional nickname of the presence's sender.

**sentpresence = None**

Flag indicating that the initial presence broadcast has been sent. Until this happens, some servers may not behave as expected when sending stanzas.

**server**

Attribute accessor for jid host

**set\_jid** (*jid*)

Rip a JID apart and claim it as our own.

**stanza = None**

A reference to `sleekxmpp.stanza` to make accessing stanza classes easier.

**start\_stream\_handler** (*xml*)

Save the stream ID once the streams have been established.

**Parameters** *xml* – The incoming stream’s root element.

**stream\_id = None**

An identifier for the stream as given by the server.

**use\_message\_ids = None**

Messages may optionally be tagged with ID values. Setting `use_message_ids` to `True` will assign all outgoing messages an ID. Some plugin features require enabling this option.

**use\_presence\_ids = None**

Presence updates may optionally be tagged with ID values. Setting `use_message_ids` to `True` will assign all outgoing messages an ID.

**username**

Attribute accessor for `jid` usernode

## 5.5 Exceptions

**exception** `sleekxmpp.exceptions.XMPPError` (*condition*=`'undefined-condition'`, *text*=`''`,  
*etype*=`'cancel'`, *extension*=`None`, *extension\_ns*=`None`, *extension\_args*=`None`, *clear*=`True`)

A generic exception that may be raised while processing an XMPP stanza to indicate that an error response stanza should be sent.

The exception method for stanza objects extending `RootStanza` will create an error stanza and initialize any additional substanzas using the extension information included in the exception.

Meant for use in SleekXMPP plugins and applications using SleekXMPP.

Extension information can be included to add additional XML elements to the generated error stanza.

### Parameters

- **condition** – The XMPP defined error condition. Defaults to `'undefined-condition'`.
- **text** – Human readable text describing the error.
- **etype** – The XMPP error type, such as `'cancel'` or `'modify'`. Defaults to `'cancel'`.
- **extension** – Tag name of the extension’s XML content.
- **extension\_ns** – XML namespace of the extensions’ XML content.
- **extension\_args** – Content and attributes for the extension element. Same as the additional arguments to the `Element` constructor.
- **clear** – Indicates if the stanza’s contents should be removed before replying with an error. Defaults to `True`.

**exception** `sleekxmpp.exceptions.IqError` (*iq*)

An exception raised when an Iq stanza of type `'error'` is received after making a blocking send call.

**iq = None**

The Iq error result stanza.

**exception** `sleekxmpp.exceptions.IqTimeout (iq)`

An exception which indicates that an IQ request response has not been received within the allotted time window.

**iq = None**

The IQ stanza whose response did not arrive before the timeout expired.

## 5.6 Jabber IDs (JID)

**class** `sleekxmpp.xmlstream.jid.JID (jid=None, **kwargs)`

A representation of a Jabber ID, or JID.

Each JID may have three components: a user, a domain, and an optional resource. For example: `user@domain/resource`

When a resource is not used, the JID is called a bare JID. The JID is a full JID otherwise.

### JID Properties:

**jid** Alias for `full`.

**full** The string value of the full JID.

**bare** The string value of the bare JID.

**user** The username portion of the JID.

**username** Alias for `user`.

**local** Alias for `user`.

**node** Alias for `user`.

**domain** The domain name portion of the JID.

**server** Alias for `domain`.

**host** Alias for `domain`.

**resource** The resource portion of the JID.

### Parameters

- **jid** (*string*) – A string of the form '`[user@]domain[/resource]`'.
- **local** (*string*) – Optional. Specify the local, or username, portion of the JID. If provided, it will override the local value provided by the *jid* parameter. The given local value will also be escaped if necessary.
- **domain** (*string*) – Optional. Specify the domain of the JID. If provided, it will override the domain given by the *jid* parameter.
- **resource** (*string*) – Optional. Specify the resource value of the JID. If provided, it will override the domain given by the *jid* parameter.

### Raises InvalidJID

**regenerate** ()

No-op Deprecated since version 1.1.10.

**reset** (*data*)

Start fresh from a new JID string.

**Parameters** `data` (*string*) – A string of the form '`[user@]domain[/resource]`'.

Deprecated since version 1.1.10.

**unescape** ()

Return an unescaped JID object.

Using an unescaped JID is preferred for displaying JIDs to humans, and they should NOT be used for any other purposes than for presentation.

**Returns** `UnescapedJID`

New in version 1.1.10.

## 5.7 Stanza Objects

The `stanzabase` module provides a wrapper for the standard `ElementTree` module that makes working with XML less painful. Instead of having to manually move up and down an element tree and insert subelements and attributes, you can interact with an object that behaves like a normal dictionary or JSON object, which silently maps keys to XML attributes and elements behind the scenes.

### 5.7.1 Overview

The usefulness of this layer grows as the XML you have to work with becomes nested. The base unit here, `ElementBase`, can map to a single XML element, or several depending on how advanced of a mapping is desired from interface keys to XML structures. For example, a single `ElementBase` derived class could easily describe:

```
<message to="user@example.com" from="friend@example.com">
  <body>Hi!</body>
  <x:extra>
    <x:item>Custom item 1</x:item>
    <x:item>Custom item 2</x:item>
    <x:item>Custom item 3</x:item>
  </x:extra>
</message>
```

If that chunk of XML were put in the `ElementBase` instance `msg`, we could extract the data from the XML using:

```
>>> msg['extra']
['Custom item 1', 'Custom item 2', 'Custom item 3']
```

Provided we set up the handler for the 'extra' interface to load the `<x:item>` element content into a list.

The key concept is that given an XML structure that will be repeatedly used, we can define a set of *interfaces* which when we read from, write to, or delete, will automatically manipulate the underlying XML as needed. In addition, some of these interfaces may in turn reference child objects which expose interfaces for particularly complex child elements of the original XML chunk.

**See Also:**

*Defining Stanza Interfaces.*

Because the `stanzabase` module was developed as part of an XMPP library, these chunks of XML are referred to as *stanzas*, and in SleekXMPP we refer to a subclass of `ElementBase` which defines the interfaces needed for interacting with a given *stanza* a *stanza object*.

To make dealing with more complicated and nested *stanzas* or XML chunks easier, *stanza objects* can be composed in two ways: as iterable child objects or as plugins. Iterable child stanzas, or *substanzas*, are accessible through a special



'substanzas' interface. This option is useful for stanzas which may contain more than one of the same kind of element. When there is only one child element, the plugin method is more useful. For plugins, a parent stanza object delegates one of its XML child elements to the plugin stanza object. Here is an example:

```
<iq type="result">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <identity category="client" type="bot" name="SleekXMPP Bot" />
  </query>
</iq>
```

We can arrange this stanza into two objects: an outer, wrapper object for dealing with the `<iq />` element and its attributes, and a plugin object to control the `<query />` payload element. If we give the plugin object the name 'disco\_info' (using its `ElementBase.plugin_attrib` value), then we can access the plugin as so:

```
>>> iq['disco_info']
'<query xmlns="http://jabber.org/protocol/disco#info">
  <identity category="client" type="bot" name="SleekXMPP Bot" />
</query>'
```

We can then drill down through the plugin object's interfaces as desired:

```
>>> iq['disco_info']['identities']
(['client', 'bot', 'SleekXMPP Bot'])
```

Plugins may also add new interfaces to the parent stanza object as if they had been defined by the parent directly, and can also override the behaviour of an interface defined by the parent.

**See Also:**

- [Creating Stanza Plugins](#)
- [Creating a Stanza Extension](#)
- [Overriding a Parent Stanza](#)

## 5.7.2 Registering Stanza Plugins

```
sleekxmpp.xmlstream.stanzabase.register_stanza_plugin(stanza, plugin,
                                                         iterable=False,
                                                         overrides=False)
```

Associate a stanza object as a plugin for another stanza.

```
>>> from sleekxmpp.xmlstream import register_stanza_plugin
>>> register_stanza_plugin(Iq, CustomStanza)
```

Plugin stanzas marked as iterable will be included in the list of `substanzas` for the parent, using `parent['substanzas']`. If the attribute `plugin_multi_attrib` was defined for the plugin, then the `substanza` set can be filtered to only instances of the plugin class. For example, given a plugin class `Foo` with `plugin_multi_attrib = 'foos'` then:

```
parent['foos']
```

would return a collection of all `Foo` `substanzas`.

### Parameters

- **stanza** (*class*) – The class of the parent stanza.
- **plugin** (*class*) – The class of the plugin stanza.

- **iterable** (*bool*) – Indicates if the plugin stanza should be included in the parent stanza’s iterable ‘substanzas’ interface results.
- **overrides** (*bool*) – Indicates if the plugin should be allowed to override the interface handlers for the parent stanza, based on the plugin’s overrides field.

New in version 1.0-Beta1: Made `register_stanza_plugin` the default name. The prior `registerStanzaPlugin` function name remains as an alias.

### 5.7.3 ElementBase

**class** `sleekxmpp.xmlstream.stanzabase.ElementBase` (*xml=None, parent=None*)

The core of SleekXMPP’s stanza XML manipulation and handling is provided by `ElementBase`. `ElementBase` wraps XML `cElementTree` objects and enables access to the XML contents through dictionary syntax, similar in style to the Ruby XMPP library `Blather`’s stanza implementation.

Stanzas are defined by their name, namespace, and interfaces. For example, a simplistic `Message` stanza could be defined as:

```
>>> class Message(ElementBase):
...     name = "message"
...     namespace = "jabber:client"
...     interfaces = set(('to', 'from', 'type', 'body'))
...     sub_interfaces = set(('body',))
```

The resulting `Message` stanza’s contents may be accessed as so:

```
>>> message['to'] = "user@example.com"
>>> message['body'] = "Hi!"
>>> message['body']
"Hi!"
>>> del message['body']
>>> message['body']
""
```

The interface values map to either custom access methods, stanza XML attributes, or (if the interface is also in `sub_interfaces`) the text contents of a stanza’s subelement.

Custom access methods may be created by adding methods of the form “`getInterface`”, “`setInterface`”, or “`delInterface`”, where “`Interface`” is the titlecase version of the interface name.

Stanzas may be extended through the use of plugins. A plugin is simply a stanza that has a `plugin_attr` value. For example:

```
>>> class MessagePlugin(ElementBase):
...     name = "custom_plugin"
...     namespace = "custom"
...     interfaces = set(('useful_thing', 'custom'))
...     plugin_attr = "custom"
```

The plugin stanza class must be associated with its intended container stanza by using `register_stanza_plugin` as so:

```
>>> register_stanza_plugin(Message, MessagePlugin)
```

The plugin may then be accessed as if it were built-in to the parent stanza:

```
>>> message['custom']['useful_thing'] = 'foo'
```

If a plugin provides an interface that is the same as the plugin's `plugin_attrib` value, then the plugin's interface may be assigned directly from the parent stanza, as shown below, but retrieving information will require all interfaces to be used, as so:

```
>>> # Same as using message['custom']['custom']
>>> message['custom'] = 'bar'
>>> # Must use all interfaces
>>> message['custom']['custom']
'bar'
```

If the plugin sets `is_extension` to `True`, then both setting and getting an interface value that is the same as the plugin's `plugin_attrib` value will work, as so:

```
>>> message['custom'] = 'bar' # Using is_extension=True
>>> message['custom']
'bar'
```

### Parameters

- **xml** – Initialize the stanza object with an existing XML object.
- **parent** – Optionally specify a parent stanza object will contain this sub stanza.

### `__bool__()`

Stanza objects should be treated as `True` in boolean contexts.

Python 3.x version.

### `__copy__()`

Return a copy of the stanza object that does not share the same underlying XML object.

### `__delitem__(attrib)`

Delete the value of a stanza interface using dict-like syntax.

Example:

```
>>> msg['body'] = "Hi!"
>>> msg['body']
'Hi!'
>>> del msg['body']
>>> msg['body']
''
```

Stanza interfaces are typically mapped directly to the underlying XML object, but can be overridden by the presence of a `del_attrib` method (or `del_foo` where the interface is named `'foo'`, etc).

The effect of deleting a stanza interface value named `foo` will be one of:

1. Call `del_foo` override handler, if it exists.
2. Call `del_foo`, if it exists.
3. Call `delFoo`, if it exists.
4. Delete `foo` element, if `'foo'` is in `sub_interfaces`.
5. Remove `foo` element if `'foo'` is in `bool_interfaces`.
6. Delete top level XML attribute named `foo`.
7. Remove the `foo` plugin, if it was loaded.
8. Do nothing.

**Parameters** *attrib* – The name of the affected stanza interface.

`__eq__` (*other*)

Compare the stanza object with another to test for equality.

Stanzas are equal if their interfaces return the same values, and if they are both instances of `ElementBase`.

**Parameters** *other* (*ElementBase*) – The stanza object to compare against.

`__getitem__` (*attrib*)

Return the value of a stanza interface using dict-like syntax.

Example:

```
>>> msg['body']
'Message contents'
```

Stanza interfaces are typically mapped directly to the underlying XML object, but can be overridden by the presence of a `get_attr` method (or `get_foo` where the interface is named `'foo'`, etc).

The search order for interface value retrieval for an interface named `'foo'` is:

- 1.The list of `substanzas` (`'substanzas'`)
- 2.The result of calling the `get_foo` override handler.
- 3.The result of calling `get_foo`.
- 4.The result of calling `getFoo`.
- 5.The contents of the `foo` subelement, if `foo` is listed in `sub_interfaces`.
- 6.True or False depending on the existence of a `foo` subelement and `foo` is in `bool_interfaces`.
- 7.The value of the `foo` attribute of the XML object.
- 8.The plugin named `'foo'`
- 9.An empty string.

**Parameters** *attrib* (*string*) – The name of the requested stanza interface.

`__iter__` ()

Return an iterator object for the stanza's `substanzas`.

The iterator is the stanza object itself. Attempting to use two iterators on the same stanza at the same time is discouraged.

`__len__` ()

Return the number of iterable `substanzas` in this stanza.

`__ne__` (*other*)

Compare the stanza object with another to test for inequality.

Stanzas are not equal if their interfaces return different values, or if they are not both instances of `ElementBase`.

**Parameters** *other* (*ElementBase*) – The stanza object to compare against.

`__next__` ()

Return the next iterable `substanza`.

`__nonzero__` ()

Stanza objects should be treated as True in boolean contexts.

Python 2.x version.

**\_\_repr\_\_** ()

Use the stanza's serialized XML as its representation.

**\_\_setitem\_\_** (*attrib*, *value*)

Set the value of a stanza interface using dictionary-like syntax.

Example:

```
>>> msg['body'] = "Hi!"
>>> msg['body']
'Hi!'
```

Stanza interfaces are typically mapped directly to the underlying XML object, but can be overridden by the presence of a `set_attr` method (or `set_foo` where the interface is named `'foo'`, etc).

The effect of interface value assignment for an interface named `'foo'` will be one of:

1. Delete the interface's contents if the value is `None`.
2. Call the `set_foo` override handler, if it exists.
3. Call `set_foo`, if it exists.
4. Call `setFoo`, if it exists.
5. Set the text of a `foo` element, if `'foo'` is in `sub_interfaces`.
6. Add or remove an empty subelement `foo` if `foo` is in `bool_interfaces`.
7. Set the value of a top level XML attribute named `foo`.
8. Attempt to pass the value to a plugin named `'foo'` using the plugin's `'foo'` interface.
9. Do nothing.

#### Parameters

- **attrib** (*string*) – The name of the stanza interface to modify.
- **value** – The new value of the stanza interface.

**\_\_str\_\_** (*top\_level\_ns=True*)

Return a string serialization of the underlying XML object.

**See Also:**

[XML Serialization](#)

**Parameters** **top\_level\_ns** (*bool*) – Display the top-most namespace. Defaults to `True`.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**\_\_delAttr** (*name*)

Remove a top level attribute of the XML object.

**Parameters** **name** – The name of the attribute.

**\_\_delSub** (*name*, *all=False*, *lang=None*)

Remove sub elements that match the given name or XPath.

If the element is in a path, then any parent elements that become empty after deleting the element may also be deleted if requested by setting `all=True`.

#### Parameters

- **name** – The name or XPath expression for the element(s) to remove.
- **all** (*bool*) – If True, remove all empty elements in the path to the deleted element. Defaults to False.

**`_del_attr`** (*name*)

Remove a top level attribute of the XML object.

**Parameters** **name** – The name of the attribute.

**`_del_sub`** (*name*, *all=False*, *lang=None*)

Remove sub elements that match the given name or XPath.

If the element is in a path, then any parent elements that become empty after deleting the element may also be deleted if requested by setting `all=True`.

**Parameters**

- **name** – The name or XPath expression for the element(s) to remove.
- **all** (*bool*) – If True, remove all empty elements in the path to the deleted element. Defaults to False.

**`_getAttr`** (*name*, *default=u''*)

Return the value of a top level attribute of the XML object.

In case the attribute has not been set, a default value can be returned instead. An empty string is returned if no other default is supplied.

**Parameters**

- **name** – The name of the attribute.
- **default** – Optional value to return if the attribute has not been set. An empty string is returned otherwise.

**`_getSubText`** (*name*, *default=u''*, *lang=None*)

Return the text contents of a sub element.

In case the element does not exist, or it has no textual content, a default value can be returned instead. An empty string is returned if no other default is supplied.

**Parameters**

- **name** – The name or XPath expression of the element.
- **default** – Optional default to return if the element does not exist. An empty string is returned otherwise.

**`_get_attr`** (*name*, *default=u''*)

Return the value of a top level attribute of the XML object.

In case the attribute has not been set, a default value can be returned instead. An empty string is returned if no other default is supplied.

**Parameters**

- **name** – The name of the attribute.
- **default** – Optional value to return if the attribute has not been set. An empty string is returned otherwise.

**`_get_stanza_values`** ()

Return A JSON/dictionary version of the XML content exposed through the stanza's interfaces:

```
>>> msg = Message()
>>> msg.values
{'body': '', 'from': , 'mucnick': '', 'mucroom': '',
'to': , 'type': 'normal', 'id': '', 'subject': ''}
```

Likewise, assigning to `values` will change the XML content:

```
>>> msg = Message()
>>> msg.values = {'body': 'Hi!', 'to': 'user@example.com'}
>>> msg
'<message to="user@example.com"><body>Hi!</body></message>'
```

New in version 1.0-Beta1.

**`_get_sub_text`** (*name*, *default=u''*, *lang=None*)

Return the text contents of a sub element.

In case the element does not exist, or it has no textual content, a default value can be returned instead. An empty string is returned if no other default is supplied.

#### Parameters

- **name** – The name or XPath expression of the element.
- **default** – Optional default to return if the element does not exist. An empty string is returned otherwise.

**`_setattr`** (*name*, *value*)

Set the value of a top level attribute of the XML object.

If the new value is None or an empty string, then the attribute will be removed.

#### Parameters

- **name** – The name of the attribute.
- **value** – The new value of the attribute, or None or '' to remove it.

**`_setSubText`** (*name*, *text=None*, *keep=False*, *lang=None*)

Set the text contents of a sub element.

In case the element does not exist, an element will be created, and its text contents will be set.

If the text is set to an empty string, or None, then the element will be removed, unless `keep` is set to True.

#### Parameters

- **name** – The name or XPath expression of the element.
- **text** – The new textual content of the element. If the text is an empty string or None, the element will be removed unless the parameter `keep` is True.
- **keep** – Indicates if the element should be kept if its text is removed. Defaults to False.

**`_set_attr`** (*name*, *value*)

Set the value of a top level attribute of the XML object.

If the new value is None or an empty string, then the attribute will be removed.

#### Parameters

- **name** – The name of the attribute.
- **value** – The new value of the attribute, or None or '' to remove it.

**\_set\_stanza\_values** (*values*)

Set multiple stanza interface values using a dictionary.

Stanza plugin values may be set using nested dictionaries.

**Parameters values** – A dictionary mapping stanza interface with values. Plugin interfaces may accept a nested dictionary that will be used recursively.

New in version 1.0-Beta1.

**\_set\_sub\_text** (*name, text=None, keep=False, lang=None*)

Set the text contents of a sub element.

In case the element does not exist, a element will be created, and its text contents will be set.

If the text is set to an empty string, or None, then the element will be removed, unless keep is set to True.

**Parameters**

- **name** – The name or XPath expression of the element.
- **text** – The new textual content of the element. If the text is an empty string or None, the element will be removed unless the parameter keep is True.
- **keep** – Indicates if the element should be kept if its text is removed. Defaults to False.

**append** (*item*)

Append either an XML object or a stanza object to this stanza object.

If a stanza object is appended, it will be added to the list of iterable stanzas.

Allows stanza objects to be used like lists.

**Parameters item** – Either an XML object or a stanza object to add to this stanza's contents.

**appendxml** (*xml*)

Append an XML object to the stanza's XML.

The added XML will not be included in the list of iterable stanzas.

**Parameters xml (XML)** – The XML object to add to the stanza.

**attrib**

Return the stanza object itself.

Older implementations of stanza objects used XML objects directly, requiring the use of `.attrib` to access attribute values.

Use of the dictionary syntax with the stanza object itself for accessing stanza interfaces is preferred. Deprecated since version 1.0.

**bool\_interfaces = set([])**

A subset of `interfaces` which maps the presence of subelements to boolean values. Using this set allows for quickly checking for the existence of empty subelements like `<required />`. New in version 1.1.

**clear** ()

Remove all XML element contents and plugins.

Any attribute values will be preserved.

**enable** (*attrib, lang=None*)

Enable and initialize a stanza plugin.

Alias for `init_plugin()`.

**Parameters attrib** (*string*) – The `plugin_attrib` value of the plugin to enable.



**find** (*xpath*)

Find an XML object in this stanza given an XPath expression.

Exposes ElementTree interface for backwards compatibility.

---

**Note:** Matching on attribute values is not supported in Python 2.6 or Python 3.1

---

**Parameters** *xpath* (*string*) – An XPath expression matching a single desired element.

**findall** (*xpath*)

Find multiple XML objects in this stanza given an XPath expression.

Exposes ElementTree interface for backwards compatibility.

---

**Note:** Matching on attribute values is not supported in Python 2.6 or Python 3.1.

---

**Parameters** *xpath* (*string*) – An XPath expression matching multiple desired elements.

**get** (*key*, *default=None*)

Return the value of a stanza interface.

If the found value is None or an empty string, return the supplied default value.

Allows stanza objects to be used like dictionaries.

**Parameters**

- **key** (*string*) – The name of the stanza interface to check.
- **default** – Value to return if the stanza interface has a value of None or "". Will default to returning None.

**getStanzaValues** ()

Return A JSON/dictionary version of the XML content exposed through the stanza's interfaces:

```
>>> msg = Message()
>>> msg.values
{'body': '', 'from': , 'mucnick': '', 'mucroom': '',
'to': , 'type': 'normal', 'id': '', 'subject': ''}
```

Likewise, assigning to *values* will change the XML content:

```
>>> msg = Message()
>>> msg.values = {'body': 'Hi!', 'to': 'user@example.com'}
>>> msg
'<message to="user@example.com"><body>Hi!</body></message>'
```

New in version 1.0-Beta1.

**initPlugin** (*attrib*, *lang=None*, *existing\_xml=None*, *reuse=True*)

Enable and initialize a stanza plugin.

**Parameters** *attrib* (*string*) – The *plugin\_attrib* value of the plugin to enable.

**init\_plugin** (*attrib*, *lang=None*, *existing\_xml=None*, *reuse=True*)

Enable and initialize a stanza plugin.

**Parameters** *attrib* (*string*) – The *plugin\_attrib* value of the plugin to enable.

**interfaces** = set([u'to', u'payload', u'type', u'id', u'from'])

The set of keys that the stanza provides for accessing and manipulating the underlying XML object. This set may be augmented with the `plugin_attrib` value of any registered stanza plugins.

**is\_extension** = False

If you need to add a new interface to an existing stanza, you can create a plugin and set `is_extension` = True. Be sure to set the `plugin_attrib` value to the desired interface name, and that it is the only interface listed in `interfaces`. Requests for the new interface from the parent stanza will be passed to the plugin directly. New in version 1.0-Beta5.

**iterables** = None

A list of child stanzas whose class is included in `plugin_iterables`.

**keys** ()

Return the names of all stanza interfaces provided by the stanza object.

Allows stanza objects to be used like dictionaries.

**lang\_interfaces** = set([])

New in version 1.1.2.

**match** (*xpath*)

Compare a stanza object with an XPath-like expression.

If the XPath matches the contents of the stanza object, the match is successful.

The XPath expression may include checks for stanza attributes. For example:

```
'presence@show=xa@priority=2/status'
```

Would match a presence stanza whose show value is set to ' xa ', has a priority value of ' 2 ', and has a status element.

**Parameters** `xpath` (*string*) – The XPath expression to check against. It may be either a string or a list of element names with attribute checks.

**name** = u'stanza'

The XML tag name of the element, not including any namespace prefixes. For example, an `ElementBase` object for `<message />` would use `name` = 'message'.

**namespace** = u'jabber:client'

The XML namespace for the element. Given `<foo xmlns="bar" />`, then `namespace` = "bar" should be used. The default namespace is `jabber:client` since this is being used in an XMPP library.

**next** ()

Return the next iterable stanza.

**overrides** = []

In some cases you may wish to override the behaviour of one of the parent stanza's interfaces. The `overrides` list specifies the interface name and access method to be overridden. For example, to override setting the parent's 'condition' interface you would use:

```
overrides = ['set_condition']
```

Getting and deleting the 'condition' interface would not be affected. New in version 1.0-Beta5.

**parent** = None

A `weakref.weakref` to the parent stanza, if there is one. If not, then `parent` is None.

**plugin\_attrib** = u'plugin'

For `ElementBase` subclasses which are intended to be used as plugins, the `plugin_attrib` value defines the plugin name. Plugins may be accessed by using the `plugin_attrib` value as the interface. An example using `plugin_attrib` = 'foo':

```
register_stanza_plugin(Message, FooPlugin)
msg = Message()
msg['foo']['an_interface_from_the_foo_plugin']
```

**plugin\_attr\_map = {}**

A mapping of the `plugin_attr` values of registered plugins to their respective classes.

**plugin\_iterables = set([])**

The set of stanza classes that can be iterated over using the ‘substanzas’ interface. Classes are added to this set when registering a plugin with `iterable=True`:

```
register_stanza_plugin(DiscoInfo, DiscoItem, iterable=True)
```

New in version 1.0-Beta5.

**plugin\_multi\_attr = u''**

For `ElementBase` subclasses that are intended to be an iterable group of items, the `plugin_multi_attr` value defines an interface for the parent stanza which returns the entire group of matching substanzas. So the following are equivalent:

```
# Given stanza class Foo, with plugin_multi_attr = 'foos'
parent['foos']
filter(isinstance(item, Foo), parent['substanzas'])
```

**plugin\_overrides = {}**

A map of interface operations to the overriding functions. For example, after overriding the `set` operation for the interface `body`, `plugin_overrides` would be:

```
{'set_body': <some function>}
```

**plugin\_tag\_map = {}**

A mapping of root element tag names (in ‘{namespace}elementname’ format) to the plugin classes responsible for them.

**plugins = None**

An ordered dictionary of plugin stanzas, mapped by their `plugin_attr` value.

**pop (index=0)**

Remove and return the last stanza in the list of iterable substanzas.

Allows stanza objects to be used like lists.

**Parameters** `index (int)` – The index of the stanza to remove.

**setStanzaValues (values)**

Set multiple stanza interface values using a dictionary.

Stanza plugin values may be set using nested dictionaries.

**Parameters** `values` – A dictionary mapping stanza interface with values. Plugin interfaces may accept a nested dictionary that will be used recursively.

New in version 1.0-Beta1.

**setup (xml=None)**

Initialize the stanza’s XML contents.

Will return `True` if XML was generated according to the stanza’s definition instead of building a stanza object from an existing XML object.

**Parameters** `xml` – An existing XML object to use for the stanza’s content instead of generating new XML.

**sub\_interfaces = set()**

A subset of `interfaces` which maps interfaces to direct subelements of the underlying XML object. Using this set, the text of these subelements may be set, retrieved, or removed without needing to define custom methods.

**subitem = set()**

A deprecated version of `plugin_iterables` that remains for backward compatibility. It required a parent stanza to know beforehand what stanza classes would be iterable:

```
class DiscoItem(ElementBase):
    ...

class DiscoInfo(ElementBase):
    subitem = (DiscoItem, )
    ...
```

Deprecated since version 1.0-Beta5.

**tag = None**

The name of the tag for the stanza's root element. It is the same as calling `tag_name()` and is formatted as `'{namespace}elementname'`.

**classmethod tag\_name()**

Return the namespaced name of the stanza's root element.

The format for the tag name is:

```
'{namespace}elementname'
```

For example, for the stanza `<foo xmlns="bar" />`, `stanza.tag_name()` would return `"{bar}foo"`.

**values**

Return A JSON/dictionary version of the XML content exposed through the stanza's interfaces:

```
>>> msg = Message()
>>> msg.values
{'body': '', 'from': , 'mucnick': '', 'mucroom': '',
'to': , 'type': 'normal', 'id': '', 'subject': ''}
```

Likewise, assigning to `values` will change the XML content:

```
>>> msg = Message()
>>> msg.values = {'body': 'Hi!', 'to': 'user@example.com'}
>>> msg
'<message to="user@example.com"><body>Hi!</body></message>'
```

New in version 1.0-Beta1.

**xml = None**

The underlying XML object for the stanza. It is a standard `xml.etree.cElementTree` object.

**xml\_ns = u'http://www.w3.org/XML/1998/namespace'**

The default XML namespace: `http://www.w3.org/XML/1998/namespace`.

## 5.7.4 StanzaBase

```
class sleekxmpp.xmlstream.stanzabase.StanzaBase(stream=None, xml=None, stype=None,
                                                sto=None, sfrom=None, sid=None, parent=None)
```

`StanzaBase` provides the foundation for all other stanza objects used by SleekXMPP, and defines a basic set

of interfaces common to nearly all stanzas. These interfaces are the `'id'`, `'type'`, `'to'`, and `'from'` attributes. An additional interface, `'payload'`, is available to access the XML contents of the stanza. Most stanza objects will provided more specific interfaces, however.

#### Stanza Interfaces:

**id** An optional id value that can be used to associate stanzas

**to** A JID object representing the recipient's JID.

**from** A JID object representing the sender's JID. with their replies.

**type** The type of stanza, typically will be `'normal'`, `'error'`, `'get'`, or `'set'`, etc.

**payload** The XML contents of the stanza.

#### Parameters

- **stream** (*XMLStream*) – Optional `sleekxmpp.xmlstream.XMLStream` object responsible for sending this stanza.
- **xml** (*XML*) – Optional XML contents to initialize stanza values.
- **stype** (*string*) – Optional stanza type value.
- **sto** – Optional string or `sleekxmpp.xmlstream.JID` object of the recipient's JID.
- **sfrom** – Optional string or `sleekxmpp.xmlstream.JID` object of the sender's JID.
- **sid** (*string*) – Optional ID value for the stanza.
- **parent** – Optionally specify a parent stanza object will contain this sub stanza.

**delPayload()**

Remove the XML contents of the stanza.

**del\_payload()**

Remove the XML contents of the stanza.

**error()**

Set the stanza's type to `'error'`.

**exception(e)**

Handle exceptions raised during stanza processing.

Meant to be overridden.

**getFrom()**

Return the value of the stanza's `'from'` attribute.

**getPayload()**

Return a list of XML objects contained in the stanza.

**getTo()**

Return the value of the stanza's `'to'` attribute.

**get\_from()**

Return the value of the stanza's `'from'` attribute.

**get\_payload()**

Return a list of XML objects contained in the stanza.

**get\_to()**

Return the value of the stanza's `'to'` attribute.

**interfaces** = set([u'to', u'payload', u'type', u'id', u'from'])

There is a small set of attributes which apply to all XMPP stanzas: the stanza type, the to and from JIDs, the stanza ID, and, especially in the case of an Iq stanza, a payload.

**namespace** = u'jabber:client'

The default XMPP client namespace

**reply** (*clear=True*)

Prepare the stanza for sending a reply.

Swaps the 'from' and 'to' attributes.

If *clear=True*, then also remove the stanza's contents to make room for the reply content.

For client streams, the 'from' attribute is removed.

**Parameters** *clear* (*bool*) – Indicates if the stanza's contents should be removed. Defaults to *True*.

**send** (*now=False*)

Queue the stanza to be sent on the XML stream.

**Parameters** *now* (*bool*) – Indicates if the queue should be skipped and the stanza sent immediately. Useful for stream initialization. Defaults to *False*.

**setFrom** (*value*)

Set the 'from' attribute of the stanza.

**Arguments:** *from* – A string or JID object representing the sender's JID.

**setPayload** (*value*)

Add XML content to the stanza.

**Parameters** *value* – Either an XML or a stanza object, or a list of XML or stanza objects.

**setTo** (*value*)

Set the 'to' attribute of the stanza.

**Parameters** *value* – A string or `sleekxmpp.xmlstream.JID` object representing the recipient's JID.

**setType** (*value*)

Set the stanza's 'type' attribute.

Only type values contained in `types` are accepted.

**Parameters** *value* (*string*) – One of the values contained in `types`

**set\_from** (*value*)

Set the 'from' attribute of the stanza.

**Arguments:** *from* – A string or JID object representing the sender's JID.

**set\_payload** (*value*)

Add XML content to the stanza.

**Parameters** *value* – Either an XML or a stanza object, or a list of XML or stanza objects.

**set\_to** (*value*)

Set the 'to' attribute of the stanza.

**Parameters** *value* – A string or `sleekxmpp.xmlstream.JID` object representing the recipient's JID.

**set\_type** (*value*)

Set the stanza's 'type' attribute.

Only type values contained in `types` are accepted.

**Parameters** *value* (*string*) – One of the values contained in `types`

**types** = set([u'set', u'unavailable', u'normal', u'get', u'chat', u'error', None])

A basic set of allowed values for the 'type' interface.

**unhandled** ()

Called if no handlers have been registered to process this stanza.

Meant to be overridden.

## 5.8 Stanza Handlers

### 5.8.1 The Basic Handler

**class** sleekxmpp.xmlstream.handler.base.**BaseHandler** (*name, matcher, stream=None*)

Base class for stream handlers. Stream handlers are matched with incoming stanzas so that the stanza may be processed in some way. Stanzas may be matched with multiple handlers.

Handler execution may take place in two phases: during the incoming stream processing, and in the main event loop. The `prerun()` method is executed in the first case, and `run()` is called during the second.

**Parameters**

- **name** (*string*) – The name of the handler.
- **matcher** – A `MatcherBase` derived object that will be used to determine if a stanza should be accepted by this handler.
- **stream** – The `XMLStream` instance that the handle will respond to.

**checkDelete** ()

Check if the handler should be removed from the list of stream handlers.

**check\_delete** ()

Check if the handler should be removed from the list of stream handlers.

**match** (*xml*)

Compare a stanza or XML object with the handler's matcher.

**Parameters** *xml* – An XML or `ElementBase` object

**name** = None

The name of the handler

**prerun** (*payload*)

Prepare the handler for execution while the XML stream is being processed.

**Parameters** *payload* – A `ElementBase` object.

**run** (*payload*)

Execute the handler after XML stream processing and during the main event loop.

**Parameters** *payload* – A `ElementBase` object.

**stream** = None

The XML stream this handler is assigned to

## 5.8.2 Callback

```
class sleekxmpp.xmlstream.handler.callback.Callback(name, matcher, pointer,
                                                    thread=False, once=False, in-
                                                    stream=False, stream=None)
```

The Callback handler will execute a callback function with matched stanzas.

The handler may execute the callback either during stream processing or during the main event loop.

Callback functions are all executed in the same thread, so be aware if you are executing functions that will block for extended periods of time. Typically, you should signal your own events using the SleekXMPP object's `event()` method to pass the stanza off to a threaded event handler for further processing.

### Parameters

- **name** (*string*) – The name of the handler.
- **matcher** – A `MatcherBase` derived object for matching stanza objects.
- **pointer** – The function to execute during callback.
- **thread** (*bool*) – **DEPRECATED**. Remains only for backwards compatibility.
- **once** (*bool*) – Indicates if the handler should be used only once. Defaults to `False`.
- **instream** (*bool*) – Indicates if the callback should be executed during stream processing instead of in the main event loop.
- **stream** – The `XMLStream` instance this handler should monitor.

**prerun** (*payload*)

Execute the callback during stream processing, if the callback was created with `instream=True`.

**Parameters** **payload** – The matched `ElementBase` object.

**run** (*payload*, *instream=False*)

Execute the callback function with the matched stanza payload.

### Parameters

- **payload** – The matched `ElementBase` object.
- **instream** (*bool*) – Force the handler to execute during stream processing. This should only be used by `prerun()`. Defaults to `False`.

## 5.8.3 Waiter

```
class sleekxmpp.xmlstream.handler.waiter.Waiter(name, matcher, stream=None)
```

The Waiter handler allows an event handler to block until a particular stanza has been received. The handler will either be given the matched stanza, or `False` if the waiter has timed out.

### Parameters

- **name** (*string*) – The name of the handler.
- **matcher** – A `MatcherBase` derived object for matching stanza objects.
- **stream** – The `XMLStream` instance this handler should monitor.

**check\_delete** ()

Always remove waiters after use.

**prerun** (*payload*)

Store the matched stanza when received during processing.



**Parameters** `payload` – The matched `ElementBase` object.

**run** (*payload*)

Do not process this handler during the main event loop.

**wait** (*timeout=None*)

Block an event handler while waiting for a stanza to arrive.

Be aware that this will impact performance if called from a non-threaded event handler.

Will return either the received stanza, or `False` if the waiter timed out.

**Parameters** `timeout` (*int*) – The number of seconds to wait for the stanza to arrive. Defaults to the stream's `response_timeout` value.

## 5.9 Stanza Matchers

### 5.9.1 The Basic Matcher

**class** `sleekxmpp.xmlstream.matcher.base.MatcherBase` (*criteria*)

Base class for stanza matchers. Stanza matchers are used to pick stanzas out of the XML stream and pass them to the appropriate stream handlers.

**Parameters** `criteria` – Object to compare some aspect of a stanza against.

**match** (*xml*)

Check if a stanza matches the stored criteria.

Meant to be overridden.

### 5.9.2 ID Matching

**class** `sleekxmpp.xmlstream.matcher.id.MatcherId` (*criteria*)

The ID matcher selects stanzas that have the same stanza 'id' interface value as the desired ID.

**match** (*xml*)

Compare the given stanza's 'id' attribute to the stored id value.

**Parameters** `xml` – The `ElementBase` stanza to compare against.

### 5.9.3 Stanza Path Matching

**class** `sleekxmpp.xmlstream.matcher.stanzapath.StanzaPath` (*criteria*)

The StanzaPath matcher selects stanzas that match a given "stanza path", which is similar to a normal XPath except that it uses the interfaces and plugins of the stanza instead of the actual, underlying XML.

**Parameters** `criteria` – Object to compare some aspect of a stanza against.

**match** (*stanza*)

Compare a stanza against a "stanza path". A stanza path is similar to an XPath expression, but uses the stanza's interfaces and plugins instead of the underlying XML. See the documentation for the stanza `match()` method for more information.

**Parameters** `stanza` – The `ElementBase` stanza to compare against.

### 5.9.4 XPath

**class** `sleekxmpp.xmlstream.matcher.xpath.MatchXPath(criteria)`

The XPath matcher selects stanzas whose XML contents matches a given XPath expression.

**Warning:** Using this matcher may not produce expected behavior when using attribute selectors. For Python 2.6 and 3.1, the `ElementTree.find()` method does not support the use of attribute selectors. If you need to support Python 2.6 or 3.1, it might be more useful to use a `StanzaPath` matcher.

If the value of `IGNORE_NS` is set to `True`, then XPath expressions will be matched without using namespaces.

**match** (*xml*)

Compare a stanza's XML contents to an XPath expression.

If the value of `IGNORE_NS` is set to `True`, then XPath expressions will be matched without using namespaces.

**Warning:** In Python 2.6 and 3.1 the `ElementTree.find()` method does not support attribute selectors in the XPath expression.

**Parameters** `xml` – The `ElementBase` stanza to compare against.

### 5.9.5 XMLMask

**class** `sleekxmpp.xmlstream.matcher.xmlmask.MatchXMLMask(criteria, default_ns='jabber:client')`

The XMLMask matcher selects stanzas whose XML matches a given XML pattern, or mask. For example, message stanzas with body elements could be matched using the mask:

```
<message xmlns="jabber:client"><body /></message>
```

Use of XMLMask is discouraged, and `MatchXPath` or `StanzaPath` should be used instead.

**Parameters** `criteria` – Either an `Element` XML object or XML string to use as a mask.

**match** (*xml*)

Compare a stanza object or XML object against the stored XML mask.

Overrides `MatcherBase.match`.

**Parameters** `xml` – The stanza object or XML object to compare against.

**setDefaultNS** (*ns*)

Set the default namespace to use during comparisons.

**Parameters** `ns` – The new namespace to use as the default.

## 5.10 XML Stream

**exception** `sleekxmpp.xmlstream.xmlstream.RestartStream`

Exception to restart stream processing, including resending the stream header.

**class** `sleekxmpp.xmlstream.xmlstream.XMLStream(socket=None, host=u'', port=0)`

An XML stream connection manager and event dispatcher.

The `XMLStream` class abstracts away the issues of establishing a connection with a server and sending and receiving XML “stanzas”. A stanza is a complete XML element that is a direct child of a root document element. Two streams are used, one for each communication direction, over the same socket. Once the connection is closed, both streams should be complete and valid XML documents.

**Three types of events are provided to manage the stream:**

**Stream** Triggered based on received stanzas, similar in concept to events in a SAX XML parser.

**Custom** Triggered manually.

**Scheduled** Triggered based on time delays.

Typically, stanzas are first processed by a stream event handler which will then trigger custom events to continue further processing, especially since custom event handlers may run in individual threads.

#### Parameters

- **socket** – Use an existing socket for the stream. Defaults to `None` to generate a new socket.
- **host** (*string*) – The name of the target server.
- **port** (*int*) – The port to use for the connection. Defaults to 0.

**add\_event\_handler** (*name, pointer, threaded=False, disposable=False*)

Add a custom event handler that will be executed whenever its event is manually triggered.

#### Parameters

- **name** – The name of the event that will trigger this handler.
- **pointer** – The function to execute.
- **threaded** – If set to `True`, the handler will execute in its own thread. Defaults to `False`.
- **disposable** – If set to `True`, the handler will be discarded after one use. Defaults to `False`.

**add\_filter** (*mode, handler, order=None*)

Add a filter for incoming or outgoing stanzas.

These filters are applied before incoming stanzas are passed to any handlers, and before outgoing stanzas are put in the send queue.

Each filter must accept a single stanza, and return either a stanza or `None`. If the filter returns `None`, then the stanza will be dropped from being processed for events or from being sent.

#### Parameters

- **mode** – One of `'in'` or `'out'`.
- **handler** – The filter function.
- **order** (*int*) – The position to insert the filter in the list of active filters.

**add\_handler** (*mask, pointer, name=None, disposable=False, threaded=False, filter=False, in-stream=False*)

A shortcut method for registering a handler using XML masks.

The use of `register_handler()` is preferred.

#### Parameters

- **mask** – An XML snippet matching the structure of the stanzas that will be passed to this handler.
- **pointer** – The handler function itself.

- **disposable** – Indicates if the handler should be discarded after one use.
- **threaded** – **DEPRECATED**. Remains for backwards compatibility.
- **filter** – **DEPRECATED**. Remains for backwards compatibility.
- **instream** – Indicates if the handler should execute during stream processing and not during normal event processing.

**Parm name** A unique name for the handler. A name will be generated if one is not provided.

**address = None**

The desired, or actual, address of the connected server.

**auto\_reconnect = None**

The `auto_reconnect` setting controls whether or not the stream will be restarted in the event of an error.

**ca\_certs = None**

Path to a file containing certificates for verifying the server SSL certificate. A non-None value will trigger certificate checking.

---

**Note:** On Mac OS X, certificates in the system keyring will be consulted, even if they are not in the provided file.

---

**certfile = None**

Path to a file containing a client certificate to use for authenticating via SASL EXTERNAL. If set, there must also be a corresponding `:attr:keyfile` value.

**configure\_dns** (*resolver, domain=None, port=None*)

Configure and set options for a `Resolver` instance, and other DNS related tasks. For example, you can also check `getaddrinfo()` to see if you need to call out to `libresolv.so.2` to run `res_init()`.

Meant to be overridden.

**Parameters**

- **resolver** – A `Resolver` instance or `None` if `dnspython` is not installed.
- **domain** – The initial domain under consideration.
- **port** – The initial port under consideration.

**configure\_socket** ()

Set timeout and other options for `self.socket`.

Meant to be overridden.

**connect** (*host='u', port=0, use\_ssl=False, use\_tls=True, reattempt=True*)

Create a new socket and connect to the server.

Setting `reattempt` to `True` will cause connection attempts to be made with an exponential backoff delay (max of `reconnect_max_delay` which defaults to 10 minute) until a successful connection is established.

**Parameters**

- **host** – The name of the desired server for the connection.
- **port** – Port to connect to on the server.
- **use\_ssl** – Flag indicating if SSL should be used by connecting directly to a port using SSL.

- **use\_tls** – Flag indicating if TLS should be used, allowing for connecting to a port without using SSL immediately and later upgrading the connection.
- **reattempt** – Flag indicating if the socket should reconnect after disconnections.

**default\_domain = None**

The domain to try when querying DNS records.

**default\_ns = None**

The default namespace of the stream content, not of the stream wrapper itself.

**default\_port = None**

The default port to return when querying DNS records.

**del\_event\_handler** (*name, pointer*)

Remove a function as a handler for an event.

#### Parameters

- **name** – The name of the event.
- **pointer** – The function to remove as a handler.

**del\_filter** (*mode, handler*)

Remove an incoming or outgoing filter.

**disconnect** (*reconnect=False, wait=None, send\_close=True*)

Terminate processing and close the XML streams.

Optionally, the connection may be reconnected and resume processing afterwards.

If the disconnect should take place after all items in the send queue have been sent, use `wait=True`.

**Warning:** If you are constantly adding items to the queue such that it is never empty, then the disconnect will not occur and the call will continue to block.

#### Parameters

- **reconnect** – Flag indicating if the connection and processing should be restarted. Defaults to `False`.
- **wait** – Flag indicating if the send queue should be emptied before disconnecting, overriding `disconnect_wait`.
- **send\_close** – Flag indicating if the stream footer should be sent before terminating the connection. Setting this to `False` prevents error loops when trying to disconnect after a socket error.

**disconnect\_wait = None**

The `disconnect_wait` setting is the default value for controlling if the system waits for the send queue to empty before ending the stream. This may be overridden by passing `wait=True` or `wait=False` to `disconnect()`. The default `disconnect_wait` value is `False`.

**dns\_answers = None**

A list of DNS results that have not yet been tried.

**dns\_service = None**

The service name to check with DNS SRV records. For example, setting this to `'xmpp-client'` would query the `_xmpp-client._tcp` service.

**end\_session\_on\_disconnect = None**

Flag for controlling if the session can be considered ended if the connection is terminated.

**event** (*name*, *data*={}, *direct*=False)

Manually trigger a custom event.

**Parameters**

- **name** – The name of the event to trigger.
- **data** – Data that will be passed to each event handler. Defaults to an empty dictionary, but is usually a stanza object.
- **direct** – Runs the event directly if True, skipping the event queue. All event handlers will run in the same thread.

**event\_handled** (*name*)

Returns the number of registered handlers for an event.

**Parameters** **name** – The name of the event to check.

**event\_queue** = None

A queue of stream, custom, and scheduled events to be processed.

**exception** (*exception*)

Process an unknown exception.

Meant to be overridden.

**Parameters** **exception** – An unhandled exception object.

**filesocket** = None

A file-like wrapper for the socket for use with the `ElementTree` module.

**getId** ()

Return the current unique stream ID in hexadecimal form.

**getNewId** ()

Generate and return a new stream ID in hexadecimal form.

Many stanzas, handlers, or matchers may require unique ID values. Using this method ensures that all new ID values are unique in this stream.

**get\_dns\_records** (*domain*, *port*=None)

Get the DNS records for a domain.

**Parameters**

- **domain** – The domain in question.
- **port** – If the results don't include a port, use this one.

**get\_id** ()

Return the current unique stream ID in hexadecimal form.

**incoming\_filter** (*xml*)

Filter incoming XML objects before they are processed.

Possible uses include remapping namespaces, or correcting elements from sources with incorrect behavior.

Meant to be overridden.

**keyfile** = None

Path to a file containing the private key for the selected client certificate to use for authenticating via SASL EXTERNAL.

**namespace\_map** = None

A mapping of XML namespaces to well-known prefixes.

**new\_id()**

Generate and return a new stream ID in hexadecimal form.

Many stanzas, handlers, or matchers may require unique ID values. Using this method ensures that all new ID values are unique in this stream.

**pick\_dns\_answer** (*domain*, *port=None*)

Pick a server and port from DNS answers.

Gets DNS answers if none available. Removes used answer from available answers.

#### Parameters

- **domain** – The domain in question.
- **port** – If the results don't include a port, use this one.

**process** (*\*\*kwargs*)

Initialize the XML streams and begin processing events.

The number of threads used for processing stream events is determined by `HANDLER_THREADS`.

#### Parameters

- **block** (*bool*) – If `False`, then event dispatcher will run in a separate thread, allowing for the stream to be used in the background for another application. Otherwise, `process(block=True)` blocks the current thread. Defaults to `False`.
- **threaded** (*bool*) – **DEPRECATED** If `True`, then event dispatcher will run in a separate thread, allowing for the stream to be used in the background for another application. Defaults to `True`. This does **not** mean that no threads are used at all if `threaded=False`.

Regardless of these threading options, these threads will always exist:

- The event queue processor
- The send queue processor
- The scheduler

**proxy\_config = None**

An optional dictionary of proxy settings. It may provide: `:host`: The host offering proxy services. `:port`: The port for the proxy service. `:username`: Optional username for accessing the proxy. `:password`: Optional password for accessing the proxy.

**reconnect** (*reattempt=True*, *wait=False*, *send\_close=True*)

Reset the stream's state and reconnect to the server.

**reconnect\_delay = None**

The current amount of time to delay attempting to reconnect. This value doubles (with some jitter) with each failed connection attempt up to `reconnect_max_delay` seconds.

**reconnect\_max\_attempts = None**

Maximum number of attempts to connect to the server before quitting and raising a 'connect\_failed' event. Setting to `None` allows infinite reattempts, while setting it to 0 will disable reconnection attempts. Defaults to `None`.

**reconnect\_max\_delay = None**

Maximum time to delay between connection attempts is one hour.

**registerHandler** (*handler*, *before=None*, *after=None*)

Add a stream event handler that will be executed when a matching stanza is received.

**Parameters** **handler** – The `BaseHandler` derived object to execute.

**registerStanza** (*stanza\_class*)

Add a stanza object class as a known root stanza.

A root stanza is one that appears as a direct child of the stream's root element.

Stanzas that appear as subanzas of a root stanza do not need to be registered here. That is done using `register_stanza_plugin()` from `sleekxmpp.xmlstream.stanzabase`.

Stanzas that are not registered will not be converted into stanza objects, but may still be processed using handlers and matchers.

**Parameters** **stanza\_class** – The top-level stanza object's class.

**register\_handler** (*handler, before=None, after=None*)

Add a stream event handler that will be executed when a matching stanza is received.

**Parameters** **handler** – The `BaseHandler` derived object to execute.

**register\_stanza** (*stanza\_class*)

Add a stanza object class as a known root stanza.

A root stanza is one that appears as a direct child of the stream's root element.

Stanzas that appear as subanzas of a root stanza do not need to be registered here. That is done using `register_stanza_plugin()` from `sleekxmpp.xmlstream.stanzabase`.

Stanzas that are not registered will not be converted into stanza objects, but may still be processed using handlers and matchers.

**Parameters** **stanza\_class** – The top-level stanza object's class.

**removeHandler** (*name*)

Remove any stream event handlers with the given name.

**Parameters** **name** – The name of the handler.

**removeStanza** (*stanza\_class*)

Remove a stanza from being a known root stanza.

A root stanza is one that appears as a direct child of the stream's root element.

Stanzas that are not registered will not be converted into stanza objects, but may still be processed using handlers and matchers.

**remove\_handler** (*name*)

Remove any stream event handlers with the given name.

**Parameters** **name** – The name of the handler.

**remove\_stanza** (*stanza\_class*)

Remove a stanza from being a known root stanza.

A root stanza is one that appears as a direct child of the stream's root element.

Stanzas that are not registered will not be converted into stanza objects, but may still be processed using handlers and matchers.

**response\_timeout = None**

The time in seconds to wait before timing out waiting for response stanzas.

**schedule** (*name, seconds, callback, args=None, kwargs=None, repeat=False*)

Schedule a callback function to execute after a given delay.

**Parameters**

- **name** – A unique name for the scheduled callback.



- **seconds** – The time in seconds to wait before executing.
- **callback** – A pointer to the function to execute.
- **args** – A tuple of arguments to pass to the function.
- **kwargs** – A dictionary of keyword arguments to pass to the function.
- **repeat** – Flag indicating if the scheduled event should be reset and repeat after executing.

**scheduler = None**

A `Scheduler` instance for executing callbacks in the future based on time delays.

**send** (*data*, *mask=None*, *timeout=None*, *now=False*, *use\_filters=True*)

A wrapper for `send_raw()` for sending stanza objects.

May optionally block until an expected response is received.

**Parameters**

- **data** – The `ElementBase` stanza to send on the stream.
- **mask** – **DEPRECATED** An XML string snippet matching the structure of the expected response. Execution will block in this thread until the response is received or a timeout occurs.
- **timeout** (*int*) – Time in seconds to wait for a response before continuing. Defaults to `response_timeout`.
- **now** (*bool*) – Indicates if the send queue should be skipped, sending the stanza immediately. Useful mainly for stream initialization stanzas. Defaults to `False`.
- **use\_filters** (*bool*) – Indicates if outgoing filters should be applied to the given stanza data. Disabling filters is useful when resending stanzas. Defaults to `True`.

**sendRaw** (*data*, *now=False*, *reconnect=None*)

Send raw data across the stream.

**Parameters**

- **data** (*string*) – Any string value.
- **reconnect** (*bool*) – Indicates if the stream should be restarted if there is an error sending the stanza. Used mainly for testing. Defaults to `auto_reconnect`.

**sendXML** (*data*, *mask=None*, *timeout=None*, *now=False*)

Send an XML object on the stream, and optionally wait for a response.

**Parameters**

- **data** – The `Element` XML object to send on the stream.
- **mask** – **DEPRECATED** An XML string snippet matching the structure of the expected response. Execution will block in this thread until the response is received or a timeout occurs.
- **timeout** (*int*) – Time in seconds to wait for a response before continuing. Defaults to `response_timeout`.
- **now** (*bool*) – Indicates if the send queue should be skipped, sending the stanza immediately. Useful mainly for stream initialization stanzas. Defaults to `False`.

**send\_queue = None**

A queue of string data to be sent over the stream.

**send\_raw** (*data*, *now=False*, *reconnect=None*)

Send raw data across the stream.

**Parameters**

- **data** (*string*) – Any string value.
- **reconnect** (*bool*) – Indicates if the stream should be restarted if there is an error sending the stanza. Used mainly for testing. Defaults to `auto_reconnect`.

**send\_xml** (*data*, *mask=None*, *timeout=None*, *now=False*)

Send an XML object on the stream, and optionally wait for a response.

**Parameters**

- **data** – The `Element` XML object to send on the stream.
- **mask** – **DEPRECATED** An XML string snippet matching the structure of the expected response. Execution will block in this thread until the response is received or a timeout occurs.
- **timeout** (*int*) – Time in seconds to wait for a response before continuing. Defaults to `response_timeout`.
- **now** (*bool*) – Indicates if the send queue should be skipped, sending the stanza immediately. Useful mainly for stream initialization stanzas. Defaults to `False`.

**session\_started\_event = None**

An `Event` to signal the start of a stream session. Until this event fires, the send queue is not used and data is sent immediately over the wire.

**session\_timeout = None**

The default time in seconds to wait for a session to start after connecting before reconnecting and trying again.

**setSocket** (*socket*, *ignore=False*)

Set the socket to use for the stream.

The filesocket will be recreated as well.

**Parameters**

- **socket** – The new socket object to use.
- **ignore** (*bool*) – If `True`, don't set the connection state to `'connected'`.

**set\_socket** (*socket*, *ignore=False*)

Set the socket to use for the stream.

The filesocket will be recreated as well.

**Parameters**

- **socket** – The new socket object to use.
- **ignore** (*bool*) – If `True`, don't set the connection state to `'connected'`.

**ssl\_retry\_delay = None**

The maximum number of times to attempt resending data due to an SSL error.

**ssl\_retry\_max = None**

The time in seconds to delay between attempts to resend data after an SSL error.

**ssl\_version = None**

Most XMPP servers support TLSv1, but OpenFire in particular does not work well with it. For OpenFire, set `ssl_version` to use SSLv23:

```
import ssl
xmpp.ssl_version = ssl.PROTOCOL_SSLv23
```

**startTLS()**  
Perform handshakes for TLS.  
If the handshake is successful, the XML stream will need to be restarted.

**start\_stream\_handler(*xml*)**  
Perform any initialization actions, such as handshakes, once the stream header has been sent.  
Meant to be overridden.

**start\_tls()**  
Perform handshakes for TLS.  
If the handshake is successful, the XML stream will need to be restarted.

**state = None**  
The connection state machine tracks if the stream is 'connected' or 'disconnected'.

**stop = None**  
An `Event` to signal that the application is stopping, and that all threads should shutdown.

**stream\_end\_event = None**  
An `Event` to signal receiving a closing stream tag from the server.

**stream\_footer = None**  
The default closing tag for the stream element.

**stream\_header = None**  
The default opening tag for the stream element.

**stream\_ns = None**  
The namespace of the enveloping stream element.

**use\_cdata = None**  
Use CDATA for escaping instead of XML entities. Defaults to `False`.

**use\_ipv6 = None**  
If set to `True`, attempt to use IPv6.

**use\_proxy = None**  
If set to `True`, attempt to connect through an HTTP proxy based on the settings in `proxy_config`.

**use\_signals(*signals=None*)**  
Register signal handlers for `SIGHUP` and `SIGTERM`.  
By using signals, a 'killed' event will be raised when the application is terminated.  
If a signal handler already existed, it will be executed first, before the 'killed' event is raised.  
**Parameters** *signals* (*list*) – A list of signal names to be monitored. Defaults to [`'SIGHUP'`, `'SIGTERM'`].

**use\_ssl = None**  
Enable connecting to the server directly over SSL, in particular when the service provides two ports: one for non-SSL traffic and another for SSL traffic.

**use\_tls = None**  
Enable connecting to the service without using SSL immediately, but allow upgrading the connection later to use SSL.

**wait\_timeout = None**

The time in seconds to wait for events from the event queue, and also the time between checks for the process stop signal.

**whitespace\_keepalive = None**

If `True`, periodically send a whitespace character over the wire to keep the connection alive. Mainly useful for connections traversing NAT.

**whitespace\_keepalive\_interval = None**

The default interval between keepalive signals when `whitespace_keepalive` is enabled.

## 5.11 Scheduler

**class** `sleekxmpp.xmlstream.scheduler.Task` (*name, seconds, callback, args=None, kwargs=None, repeat=False, qpointer=None*)

A scheduled task that will be executed by the scheduler after a given time interval has passed.

### Parameters

- **name** (*string*) – The name of the task.
- **seconds** (*int*) – The number of seconds to wait before executing.
- **callback** – The function to execute.
- **args** (*tuple*) – The arguments to pass to the callback.
- **kwargs** (*dict*) – The keyword arguments to pass to the callback.
- **repeat** (*bool*) – Indicates if the task should repeat. Defaults to `False`.
- **pointer** – A pointer to an event queue for queuing callback execution instead of executing immediately.

**args = None**

The arguments to pass to `callback`.

**callback = None**

The function to execute once enough time has passed.

**kwargs = None**

The keyword arguments to pass to `callback`.

**name = None**

The name of the task.

**next = None**

The time when the task should execute next.

**qpointer = None**

The main event queue, which allows for callbacks to be queued for execution instead of executing immediately.

**repeat = None**

Indicates if the task should repeat after executing, using the same `seconds` delay.

**reset ()**

Reset the task's timer so that it will repeat.

**run ()**

Execute the task's callback.

If an event queue was supplied, place the callback in the queue; otherwise, execute the callback immediately.

**seconds = None**

The number of seconds to wait before executing.

**class** `sleekxmpp.xmlstream.scheduler.Scheduler` (*parentstop=None*)

A threaded scheduler that allows for updates mid-execution unlike the scheduler in the standard library.

Based on: <http://docs.python.org/library/sched.html#module-sched>

**Parameters** *parentstop* – An `Event` to signal stopping the scheduler.

**add** (*name, seconds, callback, args=None, kwargs=None, repeat=False, qpointer=None*)

Schedule a new task.

**Parameters**

- **name** (*string*) – The name of the task.
- **seconds** (*int*) – The number of seconds to wait before executing.
- **callback** – The function to execute.
- **args** (*tuple*) – The arguments to pass to the callback.
- **kwargs** (*dict*) – The keyword arguments to pass to the callback.
- **repeat** (*bool*) – Indicates if the task should repeat. Defaults to `False`.
- **pointer** – A pointer to an event queue for queuing callback execution instead of executing immediately.

**addq = None**

A queue for storing tasks

**process** (*threaded=True, daemon=False*)

Begin accepting and processing scheduled tasks.

**Parameters** *threaded* (*bool*) – Indicates if the scheduler should execute in its own thread. Defaults to `True`.

**quit** ()

Shutdown the scheduler.

**remove** (*name*)

Remove a scheduled task ahead of schedule, and without executing it.

**Parameters** *name* (*string*) – The name of the task to remove.

**run = None**

A flag indicating that the scheduler is running.

**schedule = None**

A list of tasks in order of execution time.

**schedule\_lock = None**

Lock for accessing the task queue.

**stop = None**

An `Event` instance for signalling to stop the scheduler.

**thread = None**

If running in threaded mode, this will be the thread processing the schedule.

**wait\_timeout = None**

The time in seconds to wait for events from the event queue, and also the time between checks for the process stop signal.

## 5.12 XML Serialization

Since the XML layer of SleekXMPP is based on `ElementTree`, why not just use the built-in `tostring()` method? The answer is that using that method produces ugly results when using namespaces. The `tostring()` method used here intelligently hides namespaces when able and does not introduce excessive namespace prefixes:

```
>>> from sleekxmpp.xmlstream.tostring import tostring
>>> from xml.etree import cElementTree as ET
>>> xml = ET.fromstring('<foo xmlns="bar"><baz /></foo>')
>>> ET.tostring(xml)
'<ns0:foo xmlns:ns0="bar"><ns0:baz /></foo>'
>>> tostring(xml)
'<foo xmlns="bar"><baz /></foo>'
```

As a side effect of this namespace hiding, using `tostring()` may produce unexpected results depending on how the `tostring()` method is invoked. For example, when sending XML on the wire, the main XMPP stanzas with their namespace of `jabber:client` will not include the namespace because that is already declared by the stream header. But, if you create a `Message` instance and dump it to the terminal, the `jabber:client` namespace will appear.

```
sleekxmpp.xmlstream.tostring.tostring(xml=None,    xmlns=u'',    stream=None,    out-
                                     buffer=u'',    top_level=False,    open_only=False,
                                     namespaces=None)
```

Serialize an XML object to a Unicode string.

If an outer `xmlns` is provided using `xmlns`, then the current element's namespace will not be included if it matches the outer namespace. An exception is made for elements that have an attached stream, and appear at the stream root.

### Parameters

- **xml** (`Element`) – The XML object to serialize.
- **xmlns** (`string`) – Optional namespace of an element wrapping the XML object.
- **stream** (`XMLStream`) – The XML stream that generated the XML object.
- **outbuffer** (`string`) – Optional buffer for storing serializations during recursive calls.
- **top\_level** (`bool`) – Indicates that the element is the outermost element.
- **namespaces** (`set`) – Track which namespaces are in active use so that new ones can be declared when needed.

**Return type** Unicode string

### 5.12.1 Escaping Special Characters

In order to prevent errors when sending arbitrary text as the textual content of an XML element, certain characters must be escaped. These are: `&`, `<`, `>`, `"`, and `'`. The default escaping mechanism is to replace those characters with their equivalent escape entities: `&amp;`, `&lt;`, `&gt;`, `&apos;`, and `&quot;`.

In the future, the use of CDATA sections may be allowed to reduce the size of escaped text or for when other XMPP processing agents do not understand these entities.

## 5.13 Python 2.6 File Socket Shims

**class** `sleekxmpp.xmlstream.filesocket.FileSocket` (*sock*, *mode='rb'*, *bufsize=-1*,  
*close=False*)

Create a file object wrapper for a socket to work around issues present in Python 2.6 when using sockets as file objects.

The parser for `cElementTree` requires a file, but we will be reading from the XMPP connection socket instead.

**read** (*size=4096*)

Read data from the socket as if it were a file.

**class** `sleekxmpp.xmlstream.filesocket.Socket26` (*family=2*, *type=1*, *proto=0*, *\_sock=None*)

A custom socket implementation that uses our own `FileSocket` class to work around issues in Python 2.6 when using sockets as files.

**makefile** (*[mode[, bufsize]]*) → file object

Return a regular file object corresponding to the socket. The mode and bufsize arguments are as for the built-in `open()` function.

## 5.14 Core Stanzas

## 5.15 Plugins





# ADDITIONAL INFO

## 6.1 Glossary

**event handler** A callback function that responds to events raised by `XMLStream.event`. An event handler may be marked as threaded, allowing it to execute outside of the main processing loop.

**stanza object** Informally may refer both to classes which extend `ElementBase` or `StanzaBase`, and to objects of such classes.

A stanza object is a wrapper for an XML object which exposes `dict` like interfaces which may be assigned to, read from, or deleted.

**stanza plugin** A *stanza object* which has been registered as a potential child of another stanza object. The plugin stanza may accessed through the parent stanza using the plugin's `plugin_attrib` as an interface.

**stream handler** A callback function that accepts stanza objects pulled directly from the XML stream. A stream handler is encapsulated in a object that includes a *Matcher* object, and which provides additional semantics. For example, the `Waiter` handler wrapper blocks thread execution until a matching stanza is received.

**substanza** See *stanza plugin*

## 6.2 License (MIT)

Copyright (c) 2010 Nathanael C. Fritz

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## 6.2.1 Licenses of Bundled Third Party Code

### dateutil - Extensions to the standard python 2.3+ datetime module.

Copyright (c) 2003-2011 - Gustavo Niemeyer <[gustavo@niemeyer.net](mailto:gustavo@niemeyer.net)>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### fixed\_datetime

Copyright (c) 2008, Red Innovation Ltd., Finland All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Red Innovation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY RED INNOVATION “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL RED INNOVATION BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## OrderedDict - A port of the Python 2.7+ OrderedDict to Python 2.6

Copyright (c) 2009 Raymond Hettinger

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## SUELTA – A PURE-PYTHON SASL CLIENT LIBRARY

This software is subject to “The MIT License”

Copyright 2004-2013 David Alan Cridland

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## python-gnupg: A Python wrapper for the GNU Privacy Guard

Copyright (c) 2008-2012 by Vinay Sajip. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name(s) of the copyright holder(s) may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER(S) “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER(S) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### **socksipy: A Python SOCKS client module.**

Copyright 2006 Dan-Haim. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this

list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of Dan Haim nor the names of his contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY DAN HAIM “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL DAN HAIM OR HIS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- [\*License \(MIT\)\*](#)
- [\*Glossary\*](#)
- [\*genindex\*](#)
- [\*modindex\*](#)
- [\*search\*](#)

# CREDITS

**Main Author:** **Nathan Fritz** [fritzy@netflint.net](mailto:fritzy@netflint.net), [@fritzy](https://twitter.com/fritzy)

Nathan is also the author of XMPPHP and [Seesmic-AS3-XMPP](#), and a former member of the XMPP Council.

**Co-Author:** **Lance Stout** [lancestout@gmail.com](mailto:lancestout@gmail.com), [@lancestout](https://twitter.com/lancestout)

Both Fritzzy and Lance work for [&yet](#), which specializes in realtime web and XMPP applications.

- [contact@andyet.net](mailto:contact@andyet.net)
- [XMPP Consulting](#)

**Contributors:**

- Brian Beggs ([macdiesel](#))
- Dann Martens ([dannmartens](#))
- Florent Le Coz ([louiz](#))
- Kevin Smith ([Kev](#), <http://kismith.co.uk>)
- Remko Tronçon ([remko](#), <http://el-tramo.be>)
- Te-jé Rogers ([te-je](#))
- Thom Nichols ([tomstrummer](#))



# PYTHON MODULE INDEX

## S

- `sleekxmpp.basexmpp`, 66
- `sleekxmpp.clientxmpp`, 62
- `sleekxmpp.componentxmpp`, 65
- `sleekxmpp.exceptions`, 74
- `sleekxmpp.xmlstream.filesocket`, 106
- `sleekxmpp.xmlstream.handler.base`, 91
- `sleekxmpp.xmlstream.handler.callback`, 92
- `sleekxmpp.xmlstream.handler.waiter`, 92
- `sleekxmpp.xmlstream.jid`, 75
- `sleekxmpp.xmlstream.matcher.base`, 93
- `sleekxmpp.xmlstream.matcher.id`, 93
- `sleekxmpp.xmlstream.matcher.stanzapath`, 93
- `sleekxmpp.xmlstream.matcher.xmlmask`, 94
- `sleekxmpp.xmlstream.matcher.xpath`, 94
- `sleekxmpp.xmlstream.scheduler`, 104
- `sleekxmpp.xmlstream.stanzabase`, 76
- `sleekxmpp.xmlstream.tostring`, 106
- `sleekxmpp.xmlstream.xmlstream`, 94