XMPP Documentation

Release 0.1.10

Gabriel Falcao

Jan 12, 2023

Contents

Contents:

XMPP Tutorial

This is a simple barebones tutorial of XMPP in python.

Note: This tutorial **does not** cover use of parallel execution like light threads, posix threads or subprocessed. For the didatic purposes we will be building a **blocking** application.

1.1 Client TCP Connection

Let's start by creating a simple TCP connection to a XMPP server.

The XMPP toolkit provides the *XMPPConnection* that performs all the TCP socket management and exposes simple events.

Also you should never write XML manually, instead use a *XMLStream* bound to a connection in order to send

1.1.1 code

Notice the debug=True in the connection creation, that tells the lib to print the traffic in the stderr, this can be useful for debugging your application.

```python
from xmpp import XMPPConnection
from xmpp import XMLStream
from xmpp import JID


class Application(object):

    def __init__(self, jid, password):
        self.user = JID(jid)
        self.password = password
        self.connection = XMPPConnection(self.user.domain, 5222, debug=True)
```

```python
        self.stream = XMLStream(self.connection, debug=True)

        self.setup_handlers()

    def setup_handlers(self):
        self.connection.on.tcp_established(self.do_open_stream)
        self.connection.on.read(self.do_disconnect)

    def do_open_stream(self, *args, **kw):
        self.stream.open_client(self.user.domain)

    def do_disconnect(self, *args, **kw):
        self.connection.close()

    def run_forever(self):
        self.connection.connect()

        while self.connection.is_active():
            self.connection.loop_once()


if __name__ == '__main__':
    app = Application('romeo@capulet.com', 'juli3t')
    app.run_forever()
```

would output something like this

```
XMPP SEND: <?xml version='1.0'?><stream:stream
        from='romeo@capulet.com'
        to='capulet.com'
        version='1.0'
        xml:lang='en'
        xmlns='jabber:client'
        xmlns:stream='http://etherx.jabber.org/streams'>
XMPP RECV: <?xml version='1.0'?><stream:stream
      xmlns:stream='http://etherx.jabber.org/streams'
      version='1.0'
      from='capulet.com'
      id='c1a2cc21-a35d-4545-807b-2b368e567e4e'
      xml:lang='en'
      xmlns='jabber:client'>
        <stream:features>
          <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
          <register xmlns='http://jabber.org/features/iq-register'/>
          <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
            <mechanism>SCRAM-SHA-1</mechanism>
          </mechanisms>
        </stream:features>
TCP DISCONNECT: intentional
```

XMPP Connection

2.1 Events

tcp_established	the TCP connection was established
tcp_restablished	the TCP connection was lost and restablished
tcp_downgraded	the TLS connection was downgraded to TCP
tcp_disconnect	the TCP connection was lost
tcp_failed	the TCP connection failed to be established
tls_established	the TLS connection was established
tls_invalid_chain	the TLS handshake failed for invalid chain
tls_invalid_cert	the TLS handshake failed for invalid server cert
tls_failed	failed to establish a TLS connection
tls_start	started SSL negotiation
write	the TCP/TLS connection has sent data
read	the TCP/TLS connection has received data
ready_to_write	the TCP/TLS connection is ready to send data
ready_to_read	the TCP/TLS connection is ready to receive data

2.2 API

class xmpp.networking.core.**XMPPConnection**(*host,   port=5222,   debug=False, auto_reconnect=False,   queue_class=<class Queue.Queue>, hwm_in=256, hwm_out=256, recv_chunk_size=65536*)

   Event-based TCP/TLS connection.

   It buffers up received messages and also the messages to be sent.

   > **Parameters**

- **host** – a string containing a domain or ip address. If a domain is given the name will be resolved before connecting.

- **port** – defaults to `5222`. If you are using a component you might point to `5347` or something else.

- **debug** – `bool` defaults to `False`: whether to print the XML traffic on stderr

- **queue_class** – `bool` defaults to :py:class'Queue.Queue'

- **hwm_in** – `int` defaults to 256: how many incomming messages to buffer before blocking

- **hwm_out** – `int` defaults to 256: how many outcomming messages to buffer before blocking

- **recv_chunk_size** – `int` defaults to `65536`: how many bytes to read at a time.

connect(*timeout_in_seconds=3*)
> connects

>> **Parameters timeout_in_seconds** –

disconnect()
> disconencts the socket

> **published events**:

>> - `tcp_disconnect("intentional")` - when succeeded

>> **Parameters timeout_in_seconds** –

is_alive()

> **Returns** `True` if the connection is alive

loop_once(*timeout=3*)
> entrypoint for any mainloop.

> basically call this continuously to keep the connection up

perform_read(*connection*)
> reads from the socket and populates the read queue :param connection: a socket that is ready to write

perform_write(*connection*)
> consumes the write queue and writes to the given socket

>> **Parameters connection** – a socket that is ready to write

receive(*timeout=3*)
> retrieves a message from the queue, returns `None` if there are no messages.

>> **Parameters timeout** – `int` in seconds

reconnect(*timeout_in_seconds=3*)
> reconnects the socket

> **published events**:

>> - `tcp_restablished(host)` - when succeeded

>> - `tcp_failed(host)` - when failed

>> **Parameters timeout_in_seconds** –

`resolve_dns()`
> resolves the given host

`send`(*data*, *timeout=3*)
> adds bytes to the be sent in the next time the socket is ready

> > **Parameters**

> > > • **`data`** – the data to be sent

> > > • **`timeout`** – `int` in seconds

`send_whitespace_keepalive`(*timeout=3*)
> sends a whitespace keepalive to avoid connection timeouts and dead connections

> **published events**:

> > • `tcp_disconnect("intentional")` - when succeeded

> > **Parameters `timeout_in_seconds`** –

CHAPTER 3

The XML Stream

3.1 Events

feed	the XMLStream has just been fed with xml
open	the XMLStream is open
closed	the XMLStream has been closed
error	received a <stream:error></stream:error> from the server
unhandled_xml	the XMLStream failed to feed the incremental XML parser with the given value
node	a new xmpp.Node was just parsed by the stream and is available to use
iq	a new xmpp.IQ was node was received
message	a new xmpp.Message node was received
presence	a new xmpp.Presence node was received
start_stream	a new stream is being negotiated
start_tls	server sent <starttls />
tls_proceed	the peer allowed the TCP connection to upgrade to TLS
sasl_challenge	the peer sent a SASL challenge
sasl_success	the peer sent a SASL success
sasl_failure	the peer sent a SASL failure
sasl_response	the peer sent a SASL response
sasl_support	the peer says it supports SASL
bind_support	the peer says it supports binding resource
iq_result	the peer returned a <iq type-"result"></iq>
iq_set	the peer returned a <iq type-"set"></iq>
iq_get	the peer returned a <iq type-"get"></iq>
iq_error	the peer returned a <iq type-"error"></iq>
user_registration	the peer supports user registration
bound_jid	the peer returned a <jid>username@domain/resource</jid> that should be used in the from- of stanzas

3.2 API

class xmpp.stream.**XMLStream**(*connection*, *debug=False*)
XML Stream behavior class.

> **Parameters**
>
> > • **connection** – a *XMPPConnection* instance
> >
> > • **debug** – whether to print errors to the stderr

add_contact(*contact_jid*, *from_jid=None*, *groups=None*)
adds a contact to the roster of the bound_jid or the provided from_jid parameter.

Automatically sends a <presence type="subscribe"> with a subsequent <iq type="set">.

> **Parameters**
>
> > • **contact_jid** – the jid to add in the roster
> >
> > • **from_jid** – custom from= field to designate the owner of the roster
> >
> > • **groups** – a list of strings with group names to categorize this contact in the roster

bind_to_resource(*name*)
sends an <iq type="set"><resource>name</resource></iq> in order to bind the resource

> **Parameters name** – the name of the resource

bound_jid
a JID or None

Automatically captured from the XML traffic.

close(*disconnect=True*)
sends a final </stream:stream> to the server then immediately closes the bound TCP connection,disposes it and resets the minimum state kept by the stream, so it can be reutilized right away.

feed(*data*, *attempt=1*)
feeds the stream with incoming data from the XMPP server. This is the basic entrypoint for usage with the XML received from the *XMPPConnection*

> **Parameters data** – the XML string

id
returns the stream id provided by the server. <stream:stream id="SOMETHING">

Mainly used by the *authenticate()* when crafting the secret.

load_extensions()
reloads all the available extensions bound to this stream

open_client(*domain*)
Sends a <stream:stream xmlns="jabber:client"> to the given domain

> **Parameters domain** – the FQDN of the XMPP server

parse()
attempts to parse whatever is in the buffer of the incremental XML parser and creates a new parser.

ready_to_read(*_*, *connection*)
event handler for the on.ready_to_read event of a XMPP Connection.

You should probably never have to call this by hand, use bind() instead

ready_to_write(*_*, *connection*)
 even handler for the `on.ready_to_write` event of a XMPP Connection.

 You should probably never have to call this by hand, use `bind()` instead

reset()
 resets the minimal state of the XML Stream, that is: * attributes of the <stream> sent by the server during
 negotiation, used by *id()* * a bound JID sent by the server * a successful sasl result node to leverage
 `has_gone_through_sasl()`

send(*node*)
 sends a XML serialized Node through the bound XMPP connection

  Parameters node – the *Node*

send_message(*message*, *to*, ***params*)

  Parameters

   • **message** – the string with the message

   • **to** – the jid to send the message to

   • ****params** – keyword args for designating attributes of the message

send_presence(*to=None*, *delay=None*, *priority=10*, ***params*)
 sends presence

  Parameters

   • **to** – jid to receive presence.

   • **delay** – if set, it must be a ISO compatible date string

   • **priority** – the priority of this resource

send_sasl_auth(*mechanism*, *message*)
 sends a SASL response to the server in order to proceed with authentication handshakes

  Parameters mechanism – the name of SASL mechanism (i.e. SCRAM-SHA-1, PLAIN, EX-
  TERNAL)

send_sasl_response(*mechanism*, *message*)
 sends a SASL response to the server in order to proceed with authentication handshakes

  Parameters mechanism – the name of SASL mechanism (i.e. SCRAM-SHA-1, PLAIN, EX-
  TERNAL)

API Reference

class xmpp.networking.**XMPPConnection**(*host*, *port=5222*, *debug=False*, *auto_reconnect=False*, *queue_class=<class Queue.Queue>*, *hwm_in=256*, *hwm_out=256*, *recv_chunk_size=65536*)

Event-based TCP/TLS connection.

It buffers up received messages and also the messages to be sent.

> **Parameters**
>
> - **host** – a string containing a domain or ip address. If a domain is given the name will be resolved before connecting.
> - **port** – defaults to 5222. If you are using a component you might point to 5347 or something else.
> - **debug** – bool defaults to False: whether to print the XML traffic on stderr
> - **queue_class** – bool defaults to :py:class'Queue.Queue'
> - **hwm_in** – int defaults to 256: how many incomming messages to buffer before blocking
> - **hwm_out** – int defaults to 256: how many outcomming messages to buffer before blocking
> - **recv_chunk_size** – int defaults to 65536: how many bytes to read at a time.

connect(*timeout_in_seconds=3*)

> connects
>
> > **Parameters timeout_in_seconds** –

disconnect()

> disconencts the socket
>
> **published events**:
>
> - tcp_disconnect("intentional") - when succeeded
>
> > **Parameters timeout_in_seconds** –

is_alive()

> **Returns** `True` if the connection is alive

loop_once(*timeout=3*)
> entrypoint for any mainloop.
>
> basically call this continuously to keep the connection up

perform_read(*connection*)
> reads from the socket and populates the read queue :param connection: a socket that is ready to write

perform_write(*connection*)
> consumes the write queue and writes to the given socket
>
> > **Parameters** **connection** – a socket that is ready to write

receive(*timeout=3*)
> retrieves a message from the queue, returns `None` if there are no messages.
>
> > **Parameters** **timeout** – `int` in seconds

reconnect(*timeout_in_seconds=3*)
> reconnects the socket
>
> **published events**:
>
> > • `tcp_restablished(host)` - when succeeded
> >
> > • `tcp_failed(host)` - when failed
>
> > **Parameters** **timeout_in_seconds** –

resolve_dns()
> resolves the given host

send(*data*, *timeout=3*)
> adds bytes to the be sent in the next time the socket is ready
>
> > **Parameters**
> >
> > • **data** – the data to be sent
> >
> > • **timeout** – `int` in seconds

send_whitespace_keepalive(*timeout=3*)
> sends a whitespace keepalive to avoid connection timeouts and dead connections
>
> **published events**:
>
> > • `tcp_disconnect("intentional")` - when succeeded
>
> > **Parameters** **timeout_in_seconds** –

class xmpp.stream.**XMLStream**(*connection*, *debug=False*)
> XML Stream behavior class.
>
> > **Parameters**
> >
> > • **connection** – a *XMPPConnection* instance
> >
> > • **debug** – whether to print errors to the stderr

add_contact(*contact_jid*, *from_jid=None*, *groups=None*)

adds a contact to the roster of the `bound_jid` or the provided `from_jid` parameter.

Automatically sends a `<presence type="subscribe">` with a subsequent `<iq type="set">`.

Parameters

- **contact_jid** – the jid to add in the roster
- **from_jid** – custom `from=` field to designate the owner of the roster
- **groups** – a list of strings with group names to categorize this contact in the roster

bind_to_resource(*name*)

sends an `<iq type="set"><resource>name</resource></iq>` in order to bind the resource

Parameters name – the name of the resource

bound_jid

a `JID` or `None`

Automatically captured from the XML traffic.

close(*disconnect=True*)

sends a final `</stream:stream>` to the server then immediately closes the bound TCP connection,disposes it and resets the minimum state kept by the stream, so it can be reutilized right away.

feed(*data*, *attempt=1*)

feeds the stream with incoming data from the XMPP server. This is the basic entrypoint for usage with the XML received from the *XMPPConnection*

Parameters data – the XML string

id

returns the stream id provided by the server. `<stream:stream id="SOMETHING">`

Mainly used by the *authenticate()* when crafting the secret.

load_extensions()

reloads all the available extensions bound to this stream

open_client(*domain*)

Sends a <stream:stream xmlns="jabber:client"> to the given domain

Parameters domain – the FQDN of the XMPP server

parse()

attempts to parse whatever is in the buffer of the incremental XML parser and creates a new parser.

ready_to_read(*_*, *connection*)

event handler for the `on.ready_to_read` event of a XMPP Connection.

You should probably never have to call this by hand, use `bind()` instead

ready_to_write(*_*, *connection*)

even handler for the `on.ready_to_write` event of a XMPP Connection.

You should probably never have to call this by hand, use `bind()` instead

reset()

resets the minimal state of the XML Stream, that is: * attributes of the <stream> sent by the server during negotiation, used by *id()* * a bound JID sent by the server * a successful sasl result node to leverage `has_gone_through_sasl()`

send(*node*)

sends a XML serialized Node through the bound XMPP connection

> > **Parameters node** – the [*Node*]

> **send_message**(*message*, *to*, ***params*)

> > **Parameters**

> > - **message** – the string with the message
> > - **to** – the jid to send the message to
> > - ****params** – keyword args for designating attributes of the message

> **send_presence**(*to=None*, *delay=None*, *priority=10*, ***params*)
> > sends presence

> > **Parameters**

> > - **to** – jid to receive presence.
> > - **delay** – if set, it must be a ISO compatible date string
> > - **priority** – the priority of this resource

> **send_sasl_auth**(*mechanism*, *message*)
> > sends a SASL response to the server in order to proceed with authentication handshakes

> > **Parameters mechanism** – the name of SASL mechanism (i.e. SCRAM-SHA-1, PLAIN, EX-TERNAL)

> **send_sasl_response**(*mechanism*, *message*)
> > sends a SASL response to the server in order to proceed with authentication handshakes

> > **Parameters mechanism** – the name of SASL mechanism (i.e. SCRAM-SHA-1, PLAIN, EX-TERNAL)

class xmpp.models.node.**Node**(*element*, *closed=False*)
> Base class for all XML node definitions.

> The xmpp library only supports XML tags that are explicitly defined as python classes that inherit from this one.

> **classmethod create**(*_stringcontent=None*, ***kw*)
> > creates a node instance

> > **Parameters**

> > - **_stringcontent** – the content text of the tag, if any
> > - ****kw** – keyword arguments that will become tag attributes

class xmpp.models.core.**ClientStream**(*element*, *closed=False*)
> `<stream:stream xmlns='jabber:client' version="1.0" xmlns:stream='http://etherx.jabber.org/streams' />`

class xmpp.models.core.**IQ**(*element*, *closed=False*)
> `<iq></iq>`

class xmpp.models.core.**IQRegister**(*element*, *closed=False*)
> `<register xmlns="http://jabber.org/features/iq-register" />`

class xmpp.models.core.**Message**(*element*, *closed=False*)
> `<message type="chat"></message>`

exception xmpp.models.core.**MissingJID**
> raised when trying to send a stanza but it is missing either the "to" or "from" fields

---

class xmpp.models.core.**Presence**(*element*, *closed=False*)

class xmpp.models.core.**ProceedTLS**(*element*, *closed=False*)
    <proceed xmlns="urn:ietf:params:xml:ns:xmpp-tls" />

class xmpp.models.core.**SASLMechanism**(*element*, *closed=False*)

class xmpp.models.core.**SASLMechanismSet**(*element*, *closed=False*)
    <mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl"></mechanisms>

class xmpp.models.core.**StartTLS**(*element*, *closed=False*)
    <starttls xmlns="urn:ietf:params:xml:ns:xmpp-tls" />

class xmpp.models.core.**StreamFeatures**(*element*, *closed=False*)
    <stream:features></stream:features>

SASL authentication implementaion for PyXMPP.

Normative reference:

> • RFC 4422

xmpp.sasl.**filter_mechanism_list**(*mechanisms*, *properties*, *allow_insecure=False*, *server_side=False*)
    Filter a mechanisms list only to include those mechanisms that cans succeed with the provided properties and are secure enough.

> **Parameters**
>
>> • *mechanisms*: list of the mechanisms names
>>
>> • *properties*: available authentication properties
>>
>> • *allow_insecure*: allow insecure mechanisms
>
> **Types**
>
>> • *mechanisms*: sequence of *unicode*
>>
>> • *properties*: mapping
>>
>> • *allow_insecure*: *bool*
>
> **Returntype** *list* of *unicode*

xmpp.sasl.**server_authenticator_factory**(*mechanism*, *password_database*)
    Create a server authenticator object for given SASL mechanism and password databaser.

> **Parameters**
>
>> • *mechanism*: name of the SASL mechanism ("PLAIN", "DIGEST-MD5" or "GSSAPI").
>>
>> • *password_database*: name of the password database object to be used for authentication credentials verification.
>
> **Types**
>
>> • *mechanism*: *str*
>>
>> • *password_database*: *PasswordDatabase*
>
> **Raises** **KeyError** – if no server authenticator is available for this mechanism
>
> **Returns** new authenticator.
>
> **Returntype** *sasl.core.ServerAuthenticator*

xmpp.sasl.**client_authenticator_factory**(*mechanism*)

> Create a client authenticator object for given SASL mechanism.

> > **Parameters**

> > > • *mechanism*: name of the SASL mechanism ("PLAIN", "DIGEST-MD5" or "GSSAPI").

> > **Types**

> > > • *mechanism*: *unicode*

> > **Raises** **KeyError** – if no client authenticator is available for this mechanism

> > **Returns** new authenticator.

> > **Returntype** *sasl.core.ClientAuthenticator*

class xmpp.sasl.**Success**(*properties=None*, *data=None*)

> The success SASL message (sent by the server on authentication success).

class xmpp.sasl.**Failure**(*reason*)

> The failure SASL message.

> > **Ivariables**

> > > • *reason*: the failure reason.

> > **Types**

> > > • *reason*: *unicode*.

class xmpp.sasl.**Challenge**(*data*)

> The challenge SASL message (server's challenge for the client).

class xmpp.sasl.**Response**(*data*)

> The response SASL message (clients's reply the server's challenge).

class xmpp.sasl.**Reply**(*data=None*)

> Base class for SASL authentication reply objects.

> > **Ivariables**

> > > • *data*: optional reply data.

> > **Types**

> > > • *data*: *bytes*

> **encode**()

> > Base64-encode the data contained in the reply when appropriate.

> > > **Returns** encoded data.

> > > **Returntype** *unicode*

class xmpp.sasl.**PasswordDatabase**

> Password database interface.

> PasswordDatabase object is responsible for providing or verification of user authentication credentials on a server.

> All the methods of the *PasswordDatabase* may be overridden in derived classes for specific authentication and authorization policy.

> **check_password**(*username*, *password*, *properties*)

> > Check the password validity.

> > Used by plain-text authentication mechanisms.

Default implementation: retrieve a "plain" password for the *username* and *realm* using *self.get_password* and compare it with the password provided.

May be overridden e.g. to check the password against some external authentication mechanism (PAM, LDAP, etc.).

Parameters

- *username*: the username for which the password verification is requested.

- *password*: the password to verify.

- *properties*: mapping with authentication properties (those provided to the authenticator's start() method plus some already obtained via the mechanism).

Types

- *username*: *unicode*

- *password*: *unicode*

- *properties*: mapping

Returns *True* if the password is valid.

Returntype *bool*

get_password(*username*, *acceptable_formats*, *properties*)
Get the password for user authentication.

By default returns (None, None) providing no password. Should be overridden in derived classes unless only *check_password* functionality is available.

Parameters

- *username*: the username for which the password is requested.

- *acceptable_formats*: a sequence of acceptable formats of the password data. Could be "plain" (plain text password), "md5:user:realm:password" (MD5 hex digest of user:realm:password) or any other mechanism-specific encoding. This allows non-plain-text storage of passwords. But only "plain" format will work with all password authentication mechanisms.

- *properties*: mapping with authentication properties (those provided to the authenticator's start() method plus some already obtained via the mechanism).

Types

- *username*: *unicode*

- *acceptable_formats*: sequence of *unicode*

- *properties*: mapping

Returns the password and its encoding (format).

Returntype *unicode*,'unicode' tuple.

Extensions for XEPs

5.1 Service Discovery (0030)

5.1.1 Events

query_items	the server returned a list of items
query_info	the server returned a list of identities and features

5.1.2 API

class xmpp.extensions.xep0030.**ServiceDiscovery**(*stream*)

> extension for discovering information about other XMPP entities. Two kinds of information can be discovered:
> (1) the identity and capabilities of an entity, including the protocols and features it supports; and (2) the items
> associated with an entity, such as the list of rooms hosted at a multi-user chat service.

5.1.3 Example

```python
from xmpp import XMLStream
from xmpp import XMPPConnection
from xmpp import JID
from xmpp.auth import SASLAuthenticationHandler


DEBUG = True

DOMAIN = 'falcao.it'
jid = JID('presence1@falcao.it/xmpp-test')
password = 'presence1'
SASL_MECHANISM = 'SCRAM-SHA-1'
```

```python
connection = XMPPConnection(DOMAIN, 5222, debug=DEBUG)

# create a XML stream
stream = XMLStream(connection, debug=DEBUG)

# prepare the SASL mechanism
sasl = SASLAuthenticationHandler(SASL_MECHANISM, jid, password)
sasl.bind(stream)

@stream.on.closed
def stream_closed(event, node):
    connection.disconnect()
    connection.connect()
    stream.reset()

@stream.on.presence
def handle_presence(event, presence):
    logging.debug("presence from: %s %s(%s)", presence.attr['from'], presence.status.
↪strip(), presence.show.strip())

@connection.on.tcp_established
def step1_open_stream(event, host_ip):
    "sends a <stream:stream> to the XMPP server"
    logging.info("connected to %s", host_ip)
    stream.open_client(jid.domain)

@stream.on.sasl_support
def step2_send_sasl_auth(event, node):
    "sends a <auth /> to the XMPP server"
    sasl.authenticate()

@sasl.on.success
def step3_handle_success(event, result):
    "the SASL authentication succeeded, it's our time to reopen the stream"
    stream.open_client(jid.domain)

@stream.on.bind_support
def step4_bind_to_a_resource_name(event, node):
    "the server said it supports binding"
    stream.bind_to_resource(jid.resource)

@stream.on.bound_jid
def step5_send_presence(event, jid):
    stream.send_presence()
    logging.info("echobot jid: %s", jid.text)

@stream.on.presence
def step6_ensure_connectivity(event, presence):
    if presence.delay:
        stream.send_presence()

@connection.on.ready_to_write
def keep_alive(event, connection):
    if stream.has_gone_through_sasl() and (time.time() % 60 == 0):
        print 'keepalive'
        connection.send_whitespace_keepalive()

@stream.on.message
```

```python
def auto_reply(event, message):
    stream.send_presence()

    from_jid = JID(message.attr['from'])
    if message.is_composing():
        logging.warning("%s is composing", from_jid.nick)

    if message.is_active():
        logging.warning("%s is active", from_jid.nick)

    body = message.get_body()
    if body:
        logging.critical("%s says: %s", from_jid.nick, body)
        stream.send_message(body, to=from_jid.text)
        stream.send_presence(to=from_jid.text)

connection.connect()

try:
    while connection.is_alive():
        connection.loop_once()

except KeyboardInterrupt as e:
    print "\r{0}".format(traceback.format_exc(e))

    raise SystemExit(1)
```

5.2 Component (0114)

5.2.1 Events

success	the server sent a `<handshake />`
error	the server returned a `<stream:error>`

5.2.2 API

class xmpp.extensions.xep0114.**Component**(*stream*)

Provides an external component API while keeping minimal state based on a single boolean flag.

authenticate(*secret*)

sends a `<handshake>` to the server with the encoded version of the given secret :param secret: the secret string to authenticate the component

create_node(*to*, *tls=False*)

creates a `ComponentStream` with an optional `<starttls />` in it.

is_authenticated()

> **Returns** `True` if a success handshake was received by the bound

XMLStream

open(*domain*, *tls=False*)

sends an `<stream:stream xmlns="jabber:component:accept">`

5.3 Create your own

You can easily have your own implementation of a XEP by extending the class `xmpp.extensions.Extension`.

As long as your implementation is being imported by your application, the XMPP toolkit will automatically recognize your subclass and make it available whenever a `XMPPStream` is instantiated.

5.3.1 XEP 9999

Let's come up with our own XEP

1. Introduction

This document defines a protocol for communicating *dummy* from one user to another. Such information MUST be appended to a `received_dummy_list` in the *receiving* entity. The entity MAY also send a *dummy* which SHALL be appended to a `sent_dummy_list` in the *sending* entity.

2. Protocol

Sending a dummy

```
<iq id="23713d" type="set" from="tybalt@shakespeare.org" to="rosaline@shakespeare.org
↪">
  <dummy xmlns="xmpp:xep:example">Romeo</dummy>
</iq>
```

Receiving a dummy

```
<iq id="23713d" type="result" from="tybalt@shakespeare.org" to="rosaline@shakespeare.
↪org">
  <dummy xmlns="xmpp:xep:example">Juliet</dummy>
</iq>
```

Here is the implementation, notice its statelessness

```python
from speakers import Speaker as Events
from xmpp.models import Node, IQ, JID
from xmpp.extensions import Extension


class Dummy(Node):
    __tag__ = 'dummy'
    __etag__ = '{xmpp:xep:example}dummy'
    __namespaces__ = [
        ('', 'xmpp:xep:example')
    ]
    __children_of__ = IQ


class Fake(Extension):
    __xep__ = '9999'

    def initialize(self):
        self.on = Events('fake', [
```

```
            'dummy',  # the server sent a dummy inside of an IQ
        ])
        self.stream.on.node(self.route_nodes)

    def route_nodes(self, _, node):
        if isinstance(node, Dummy):
            self.on.dummy.shout(node)


    def send_dummy(self, to, value):
        params = {
            'to': to,
            'type': 'set',
        }
        node = IQ.with_child_and_attributes(
            Dummy.create(value),
            **params
        )
        self.stream.send(node)
```

Usage of your newly created extension

```python
from xmpp import XMLStream
from xmpp import XMPPConnection
from xmpp import JID
from xmpp.auth import SASLAuthenticationHandler


DEBUG = True


DOMAIN = 'shakespeare.oreg'
jid = JID('tybalt@shakespeare.oef/cahoots')
password = 'sk3tchy'


SASL_MECHANISM = 'SCRAM-SHA-1'



RECEIVED_DUMMY_LIST = []
SENT_DUMMY_LIST = []


connection = XMPPConnection(DOMAIN, 5222, debug=DEBUG)
stream = XMLStream(connection, debug=DEBUG)


sasl = SASLAuthenticationHandler(SASL_MECHANISM, jid, password)
sasl.bind(stream)


@connection.on.tcp_established
def step1_open_stream(event, host_ip):
    stream.open_client(jid.domain)


@stream.on.sasl_support
def step2_send_sasl_auth(event, node):
    sasl.authenticate()


@sasl.on.success
def step3_handle_success(event, result):
    stream.open_client(jid.domain)


@stream.on.bind_support
def step4_bind_to_a_resource_name(event, node):
```

```python
        stream.bind_to_resource(jid.resource)

@stream.on.bound_jid
def step5_send_presence(event, jid):
    dummies.send_dummy(to='rosaline@shakespeare.org', value='Romeo')
    SENT_DUMMY_LIST.append('Romeo')

@dummies.on.dummy
def step6_store_dummy(event, dummy):
    RECEIVED_DUMMY_LIST.append(dummy.value)


connection.connect()

try:
    while connection.is_alive():
        connection.loop_once()

except KeyboardInterrupt as e:
    print "\r{0}".format(traceback.format_exc(e))

    raise SystemExit(1)
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index