OTR bot Documentation

Release 1.0 Alpha

Chris Snijder, Maarten de Waard

Aug 15, 2017

Table of Contents

The bot exists of four modules:

- The *OTR Bot* module that runs an XMPP client that communicates with peers, to teach them how to enable and verify OTR connections.
- The *OTR Plugin* module that enables the use of OTR with SleekXMPP.
- A *State Machine* that executes actions and handles events.
- A Learning Tools Interoperability (LTI) interface to be able to start talking with the OTR bot from a Learning Management System (LMS).

CHAPTER 1

Quick Start

- It is recommended to start by creating a *virtualenv* environment for the bot. The bot uses external python libraries that are fixed on a certain version. The easiest way to ensure that nothing else on your system breaks when installing these dependencies, is by using a virtual environment.
- Install python-dev in order to be able to install all dependencies. On debian-based systems, run *apt-get install python-dev*.
- Run *pip install -r requirements.txt* to fulfill all python requirements.
- By default, the environment in *settings/env.py* is set to production. This means it searches for *production.py* in the settings folder. Alter this environment, or the *production.py* file:
  - The *XMPP_ACCOUNTS* dictionary should at least contain one valid jabber account. Its JID should be the key, its value a dictionary with the contents 'password', 'private_key', 'SSL', 'port' and 'allow_plain_text'.
  - Enter the JID of one of the accounts in *XMPP_ACCOUNTS* in *XMPP_DEFAULT_ACCOUNTS*. This is the default account to start the bot with. If you don't have a default account, always start the bot with the –jid flag.
- Start the bot by running *./bot.py runclient*. Run *./bot.py runclient –help* for information on the available arguments. Other subcommands are available as well. Run *python bot.py –help* for more information.

OTR Bot

XMPP client for the OTR bot. Manages all conversations.

Usage

Start a bot by initialising the class:

```
>>> bot = OtrBot()
```

Then, you can connect to the Jabber server:

```
>>> bot.connect()
```

Lastly, start the bot by running `process`:

```
>>> bot.process()
```

If a user wants to be able to talk to the OTR bot, you should be added to its whitelist. This is achieved by running *OtrBot.add_jid()*, or adding the jid to the whitelist in the *Settings*. When adding someone to the whitelist, it is possible to supply its locale. All supported locales are in the root directory, in the locales folder.

After running add_jid, the bot will add a user to his whitelist and roster, and will invite the user to talk with him over Jabber. When the user starts talking to the bot, it will start a session for the user, and respond to him. When a user is inactive for too long, his session will be removed from the bot. See the functions *warn_session()*, *kill_session()* and *terminate_session()* for more information.

The bot is supplied in conjunction with an LTI interface for adding Jabber IDs to the bot and supplying shared secrets, but it is possible to use the bot from the command line, without using the LTI interface.

SleekXMPP

The bot uses SleekXMPP for communication and scheduling. A custom *OTR Plugin* fires events through SleekXMPP and are used by the bot to react on the user's actions.

Class documentation

class otrbot.core.client.**OtrBot**(*jid=None*, ***kwargs*)

OTR bot ClientXMPP implementation

Start an OTR bot for a specifig Jabber ID.

Parameters

- **jid** (*str*) – The Jabber ID the bot can use. Its password needs to be in the settings, or supplied as a keyword argument. :keyword str password: The password for the jabber account
- **default_locale** (*str*) – The default locale for the bot, overrides the setting DEFAULT_LOCALE.
- **shared_secret** (*str*) – The default shared secret for the bot, overrides the setting DEFAULT_SECRET. Can be None, if the secret is added via another route, e.g., LTI.
- **key_file** (*str*) – The name of the file containing the private key for the bot

REFRESH_SESSION_WHITELIST = ('message_received', 'otr_enabled', 'smp_started', 'smp_aborted', 'otr_disabled')

The events that allow the session of a user to be refreshed, extending its timeout

SIGN_OUT_MESSAGE = 'Your session has expired, you can start a new session from ${project_name}.'

This message is sent when the bot signs out and no other message is supplied

connect()

Connecting with the TLS and SSL flags set to None allows manipulation of these settings through the XML stream object

bot_connected(*event*)

Verify that the connection is established as requested, i.e.: the TLS cipher(s) that was(/were) configured is(/are) used. If this is not the case, disconnect and quit with a fatal error.

session_start(*event*)

Bring the bot online. All the JIDs that are already in the bot's whitelist are added to the roster.

session_end(*event*)

Clean up at the end of the session. Kill all sessions, delete all aggregated data and empty the roster.

otr_event (*event*)
Apply an OTR event to the state machine of a specific JID. OTR events are fired by the *OtrPlugin*.

If the JID was unknown in the _sessions dict, but is in the _whitelist, a new session is started with start_sessions.

Parameters **event** (*dict*) – A dictionary containing the keys jid (JID), type (str) and data (any type). The type should be an event that can be handled by the current state. If the event can not be handled by the state, it will raise a CannotHandle exception.

check_sessions ()
Checks all sessions' contexts in self._sessions for timeouts. If a timeout is imminent (WARNING_TIMEOUT seconds have passed), it fires a warning using *warn_session()*. If a timeout has passed (SESSION_TIMEOUT seconds have passed), it removes the session by using *kill_session()*.

refresh_session (*jid*)
Refresh the users session by resetting the timeouts. If the user has been warned about a session timeout, the bot sends a notification that the session has been refreshed.

Parameters **jid** (*JID*) – The user's jid.

warn_session (*jid*)
Send a warning to the user, that they should reply to maintain the session.

Parameters **jid** (*JID*) – The user's jid.

kill_session (*jid*, *msg=None*)
Tell the user the session has expired, then set a timeout for termination of the session (allows the message to be sent out first).

Parameters

- **jid** (*JID*) – The user's jid, an instance of sleekxmpp.jid.JID
- **msg** (*str*) – The message that will be sent by the bot before going down.

terminate_session (*jid*, *msg=None*)
Remove the session from _sessions and the user from the roster. Also makes the bot offline to the user. A status message is set explaining how the user can re-enable the bot.

Parameters

- **jid** (*JID*) – The user's jid, an instance of sleekxmpp.jid.JID
- **msg** (*str*) – The status message for the bot the user will see.

changed_subscription (*presence*)
Only authorize and subscribe to people in self._whitelist, which is populated by settings.JID_WHITELIST.

Parameters **presence** (*Presence*) – A sleekxmpp.stanza.presence.Presence stanza for the subscription

add_jid (*jid*, *locale='en_GB'*)
Add the supplied JID to the whitelist and the roster and send a presence to appear online for that user.

Parameters **jid** (*JID*) – The Jabber ID to add

get_shared_secret (*jid*)
Return the shared secret set by the jabber ID jid, or none if he has not set one

Parameters **jid** (*str*) – The JID that should have set a shared secret

set_shared_secret (*jid*, *secret*)
Insert the secret into the context for the jid. The otr_event 'secret_received' is fired afterwards.

Parameters

- **jid** (*str*) – A valid jabber ID that is in self._sessions
- **secret** (*str*) – A secret that should be supplied by the user

check_bot_secret (*jid*, *secret*)
Check if the secret that the bot has provided via chat corresponds with the secret that is inserted into this function

Parameters

- **jid** (*str*) – The JID that received the secret
- **secret** (*str*) – The secret. Note that this is **not** the SMP shared secret

Return bool True if the secrets are the same, False otherwise.

Raises **JidNotKnown** – If the jid is not in _sessions, an exception is raised.

jid_in_sessions (*jid*)
Checks if the supplied JID has a running session.

Parameters **jid** (*str*) – The Jabber ID

Returns bool True if the jid is a key in self._sessions

Translations

The OTR bot supports translations made with Gettext. The translation files are placed in the `otrbot/locales/<language>` directories. The otrbot comes with four commands that help you setup new translations and update existing ones. run `./bot -h` to get an overview of the commands. Each subcommand also supports help, so when in doubt, just try `./bot.py <subcommand> -h`

Extracting translation messages

When the otr bot's messages have been altered, or new ones have been added, the command `./bot extract` will extract these new messages from the program code. This generates a new `.pot` file (`messages.pot` by default) that can be used to update the current translations. If you want the `.pot` file to have another name, supply that name with the `-o` flag.

Starting a translation

To start a translation, you use the `init` subcommand. When you run `./bot init <pot-file> <locale>`, a new locale will be created in the `otrbot/locales` directory. So, let's say you want to start an en_US translation and you extracted the translations to `messages.pot`, run `./bot init messages.pot en_US`.

Updating an existing translation

When you have an updated `.pot` file, because the application changed, you can update your translation files by running `./bot update <pot-file> <locale>`. If you used the default filename (`messages.pot`) for extraction and, for example, want to update your en_US translation, you run `./bot update messages.pot en_US`. The program will notify you of which file it created. You can then edit that file with your favorite editor.

Compiling your changes

When you are happy with your changes, you should compile the new messages to a messages.mo file, a binary file that can be read quickly by the computer. To compile your locale files, run `./bot compile`. This automatically compiles all the locales in the `otrbot/locales` directory. If you only want to compile one of the locales, you can use the `-l` flag. For example, to only compile your en_GB translation, run `./bot compile -l en_US`.

OTR Plugin

SleekXMPP Plugin to use OTR

Uses Python-Potr in combination with SleekXMPP to enable OTR messaging for Jabber clients. This plugin adds the event 'message_received', which forwards decrypted messages as well. Use that instead of the 'message' event. Furthermore, the plugin fires 'otr_event' events. These events always consist of a dictionary with the following content (keys):

jid The Jabber ID this event is about (This is a SleekXMPP JID object)

type The name of the event type that is fired, e.g. "otr_enabled"

data Accompanying data for the event, this may be of any class, but we would recommend using strings. Currently, no events use this field.

List of event types

otr_enabled OTR was off and is now on

otr_disabled OTR was on and is now off

otr_already_setup OTR conversation was finished and is refreshed

smp_started User tried to start SMP

smp_question_used User tried to start SMP with a question

smp_aborted SMP was aborted from either side.

correct_secret_received User supplied the secret through anothes channel and was received correctly. SMP trust is now established.

incorrect_secret_received No trust has been established after SMP

unsafe_secret_received Not implemented in this plugin. Can be used if you have a way to check the strength of secrets. Look at `core.otrbot.state.action_check_secret_safe()` for an example

Credits

The following two scripts have been used as a basis for this plugin:

- https://gist.github.com/marianoguerra/4023941
- https://github.com/mikegogulski/python-otrxmppchannel

class `otrbot.otrplugin.plugin.`**`OtrPlugin`**(*xmpp*, *config=None*)
OtrPlugin: enables OTR commands

plugin_init()
Initialise plugin. Sets the description (needed for being a SleekXMPP plugin) to the docstring of this class. Makes this plugin handle the 'message' event and registers the OtrMessage stanza plugin.

setup_otr(*xmpp_account*)
Setup OTR account to be used for encryption and decryption. This function should be run after plugin_init and before anything else.

Parameters **xmpp_account** (*dict*) – An account as found in the sample settings. This needs at least a JID. Private keys for the account will be generated automatically

set_secret(*jid*, *secret*, *start_smp=False*)
Set the secret in the otr_context for this jid to this secret. Create a context if it does not exist for this jid.

Parameters

- **jid** (*str*) – The jabber ID to set the secret for
- **secret** (*str*) – The secret that should be set.
- **start_smp** (*bool*) – If this is true, smpGotSecret is run in the OtrContext. If SMP has not started yet, this means that the bot will start it.

decrypt(*message*)
Try to decrypt a message and sets message.otr_state to a state from OTR_STATES. Then launches the event 'message_received', unless the message was empty.

Parameters **message** (*str*) – A string that can be decrypted by Python-Potr

encrypt(*jid*, *message_string*)
Send message "message" to "jid" if encryption is possible. If there's no encryption, "jid" is asked to enable encryption(?)

Parameters

- **jid** (*str*) – a JID to whom the message will be sent
- **message_string** (*str*) – A message to send

Returns The message as it was sent

emit_event(*jid*, *event*, *data=None*)
Emit an otr_event on xmpp

Parameters

- **jid** (*str*) – The JID that the event is for/started by
- **event** (*str*) – The type of the event
- **data** – accompanying data. For example the event 'message_received' has the message body in the data variable. This variable may be of any data type.

send_plain(*jid*, *message_string*)
Send a message over xmpp without checking anything. The :class OtrContext uses this to inject messages

Parameters

- **jid** (*JID*) – The Jabber ID to send to
- **message_string** (*str*) – The message to send

Otr Context that keeps all the otr-relevant information for a certain peer

class otrbot.otrplugin.context.**OtrContext**(*account*, *otr_plugin*, *peer*, *fingerprint*)
Knows how peers can be contacted (OTR enabled or not, etc.) Implements these functions from Context:

- getPolicy
- setState
- inject

Create a context for the account to OTR-message with.

Parameters

- **account** (*OtrAccount*) – The OTR account
- **otr_plugin** (*OtrPlugin*) – An OTR plugin instance
- **peer** (*str*) – A JID
- **fingerprint** (*str*) – fingerprint for the JID, becomes lower case, is allowed to be None

emit_event(*event*, *data=None*)

Makes the otr_plugin emit an event for self.peer

Parameters

- **event** (*str*) – An otr_event type that can be executed by the OtrBot.
- **data** (*str*) – The data of the event, e.g., a message for a message_received event.

processTLVs(*tlvs*, *appdata=None*)

Process typle/length/value records. First check if message is an SMP message and fire the appropriate otr_event if so. Then call the super to do further processing.

Parameters

- **tlvs** (*list*) – a list of TLV objects
- **appdata** – Gets passed to super

getPolicy(*key*)

Returns the default policy from DEFAULT_POLICY_FLAGS. Returns False if the policy does not exist.

Parameters **key** – The policy that should be returned. Possible keys:

- ALLOW_V1
- ALLOW_V2
- REQUIRE_ENCRYPTION
- SEND_TAG
- WHITESPACE_START_AKE
- ERROR_START_AKE

setState(*new_state*)

Sets trust based on their fingerprint

Parameters **new_state** (*int*) – The new state corresponding to the possible states in potr.context.

inject(*msg*, *appdata=None*)

Use OtrPlugin to send a message. This will be encrypted if possible.

Parameters

- **msg** (*str*) – The message that will be injected.
- **appdata** (*dict*) – This is not used.

smpAbort(*appdata=None*)
This function is overriden in order to fire an "smp_aborted" otr_event when it is called.

Parameters **appdata** (*dict*) – This is forwarded to other calls.

Simple implementation of the abstract potr Account class.

class otrbot.otrplugin.account.**OtrAccount**(*otr_plugin*, *jid*, *private_key=None*)
Implementation of potr Account class

Taken from OtrXMPPChannel

Init and create a private key if needed

loadPrivkey()
overload not-implemented load function

savePrivkey()
Should be overloaded, but don't know with what functionality

saveTrusts()
Should be overloaded, but don't know with what functionality. Apparently this function is called after a user ends a private conversation

Simple implementation of the abstract potr Manager class.

class otrbot.otrplugin.manager.**OtrManager**(*jid*, *otr_plugin*)
Class that retrieves contexts for messages. The contexts manage decryption and encryption of messages

Initialise with account and empty contexts dictionary. The dictionary wil contain bare JIDs and OtrContext objects. Each JID will talk to a different OtrAccount instance of the bot. This way, the bot always has a different fingerprint for each session, enabling a user to go through the same steps several times.

contexts = None
Dictionary of people the bot is talking to - Keys: str of bare jids - Values: OtrContexts objects

get_context(*jid*)
Return the context for the conversation with JID jid. If it does not exist, a new account (with a new random private key) is created for the session with the jid.

Parameters

- **jid** (*str*) – the Jabber ID of the peer
- **fingerprint** (*str*) – An optional fingerprint of the JID for trying to manually establish trust (not tested).

destroy_context(*jid*)
Remove the context with JID jid from the contexts

Parameters **jid** (*JID*) –

destroy_all_contexts()
Remove all context for converations

State Machine

State Machine module

This module contains the *StateMachine* class.

This *StateMachine* class is designed to do 3 specific things:

- Allow only transitions that are valid for the current state (enforced by the `State` object).
- Pass events to the `State` and let the state handle them. You can pass both transition events as well as events that should trigger something, which you can define yourself. The `State` class can be extended with `handle_...()` functions that handle these specific events. You should be careful not to name your transitions the same as your `handle_...()` functions because an exception will be raised if they collide.
- Pass actions to the `State` and let the state handle them. The `State` class can be extended with `action_...()` functions that do these specific actions.

The state machine's `__init__()` takes a dict argument that contains all states the state machine should support.

The states can have 'on_enter' and 'on_leave' definitions, which are lists of actions to do when the state is entered or left respectively. These actions have to be defined in the `State` object for this to work. There should also be a list of 'allowed_actions' array defined in each state definition, this array should contain the action names without the prefix 'action_'. The reason the 'action_' is not prefixed is that the prefix string can be changed in a class extension, to e.g.: 'do_'.

class `otrbot.statemachine.statemachine.`**StateMachine**(*states_object*, *ctxt*)

This class in not supposed to be used directly, you should extend this class and add the application specific functions to it.

You will most likely require your own definition of the `State` class as well, to override the definition of the `State` class the state machine uses, you will need to override the *STATE_OBJECT* in your state machine class extension.

You should also use `super()` to initialise the *StateMachine* object, you will need to pass some arguments too, see below.

Initialise the *StateMachine* class.

Parameters

- **states_object** (*dict*) – A dict containing all possible states, its transitions, actions, 'on_enter' and 'on_leave' actions
- **ctxt** (`SharedContext`) – A shared context that contains the current state context data

Raises **StateDoesNotExist** – Indicates that the state you chose to be the initial state is not in the states_object

STATE_OBJECT

alias of `State`

action(*action*, **args*, ***kwargs*)

Run an action using the current state.

Parameters

- **action** (*str*) – String containing the function name of the action you want to execute
- **args** (*set*) – Arbitrary positional arguments to pass to the action
- **kwargs** (*dict*) – Arbitrary keyword arguments to pass to the action

The action will be handled by the current state which is an instance of `State`. See `State.action()` for more information.

state

Return current state.

Returns State The instance of the current `State` object.

handle(*event*, **args*, ***kwargs*)

Handle an event using the current state.

Parameters

- **event** (*str*) – String containing the event name of the event you want handled by the state.
- **args** (*set*) – Arbitrary positional arguments to pass to the action
- **kwargs** (*dict*) – Arbitrary keyword arguments to pass to the action

The event will be handled by the current state which is an instance of `State`. See `State.handle()` for more information.

ctxt
Return the current context.

Returns SharedContext The shared context used by the state machine

State module

This module contains the *State* class for the `StateMachine`.

You can extend this class to add `action_...()` and `handle_...()` functions. To let the `StateMachine` use your own custom state class, you need to extend the `StateMachine` class and override the `StateMachine.STATE_OBJECT` constant.

class `otrbot.statemachine.state.`**State**(*name*, *states_object*, *ctxt*)
This class in not supposed to be used directly, you should extend this class and add the application specific functions to it.

To override the definition of the *State* classthe state machine uses, you will need to override the `STATE_OBJECT` in your state machine class extension.

You should also use `super()` to initialise the *State* object, you will need to pass some arguments too, see below.

If you want to define different prefixes for the `action_...()` or `handle_...()` functions, e.g.: `do_...()`, you can override the `PREFIX_ACTION` or the `PREFIX_HANDLE` class constants.

Initialise the state.

Parameters

- **name** (*str*) – The name of the current state
- **states_object** (*dict*) – A dict containing all possible states, its transitions, actions, 'on_enter' and 'on_leave' actions
- **ctxt** (`SharedContext`) – A shared context that contains the current state context data

on_enter()
When the state is entered the "on_enter" actions will be executed.

on_leave()
When the state is left the "on_leave" actions will be executed.

action(*action*, **args*, ***kwargs*)
Run an action that is defined in the *State* and prefixed by `PREFIX_ACTION`.

Parameters

- **action** (*str*) – Name of the action to run (without prefix)
- **args** (*set*) – Arbitrary positional arguments to pass to the action
- **kwargs** (*dict*) – Arbitrary keyword arguments to pass to the action

Raises

- **NotImplementedError** – If the requested action is not implemented
- **ActionNotAllowed** – If the action is not allowed within the current state.

handle (*event*, **args*, ***kwargs*)
Handle events passed to the *State* object.

There are 2 types of events that can be handled by state objects:

•Transition events, which should cause a transition to another state if the transition is allowed.

•Events that should be handled by the state without transitioning.

Parameters

- **event** (*str*) – Event name.
- **args** (*set*) – Arbitrary positional arguments to pass to the action
- **kwargs** (*dict*) – Arbitrary keyword arguments to pass to the action

Raises **CannotHandle** – If there is no way to handle the event.

possible_actions
Return all actions that are defined in the *State* object.

The results of the introspection are cached, if you dynamically modify the object after getting this property the changes will not be reflected by the result.

Returns set All actions defined in this *State* object.

transitions
Return transition

Returns dict All transitions supported by this *State* object.

ctxt
Return the current context.

Returns SharedContext The shared context used by the state object

Shared Context module

This module contains the *SharedContext* class.

This *SharedContext* hold all the data that needs to be shared between the `State` and the `StateMachine` objects.

class `otrbot.statemachine.context.`**SharedContext** (*state=None*)
A shared context for the state machine and the state.

Parameters **state** (*str*) – The curent StateMachine state

LTI

Adding the bot to your LMS

To add this tool to your LMS, you need to run it on your own server, let's say example.com. Currently it is started by running ./bot.py runclient, which starts an OTR bot and the LTI server. The server can then be reached at exam-

ple.com:5000.

In your LMS, enter example.com:5000 as the launch URL for this LTI application. The "Consumer key" and "Shared secret" are in the settings file for your environment. In our `settings.development_sample` settings, find *PYLTI_CONFIG*. You can see that we added one consumer under "consumers", which has `__consumer_key__` as his consumer key, and `__lti_secret__` as his secret. It is recommended to change these defaults.

Now it's time to add your bot to the plugins/apps in your LMS.

Open EdX

To add your bot to a course in Open EdX, follow these steps.

1. In Open EdX studio, go to *Settings* -> *Advanced Settings*
2. In *Advanced Module List*, enable the LTI module, by adding `"lti"` to the list. If you have no other modules enabled, the value of the text field will look like `[ "lti" ]`.
3. In *LTI Passports*, add the otrbot. This is done by adding three colon (:) separated values to the list, in 1 string: `"otrbot:<consumer_key>:<lti_secret>"`. Make sure that you fill in `<consumer_key>` and `<lti_secret>` with the same values as you entered in the settings file.

Now, in your course:

1. Add a unit, you will see that you can choose an "Advanced" module. Click that, and choose *LTI*.
2. Click the edit button of the LTI module. Here, fill in
   - **LTI URL**: example.com:5000 (replace example.com with the URL to your OTR bot machine).
   - **LTI ID**: otrbot
   - Take a look at the other settings. You might want to configure *Open in New Page* and *Display Name* as well.

You should now be able to use the LTI otr-bot component

Canvas

In canvas, follow these steps to add the bot to a course:

- In your course, click Settings and then go to the Apps tab.
- Click Add App
- Choose "Manual entry"
- The only fields that are currently relevant in Canvas are "Consumer key", "Shared secret" and "Launch URL". Follow your previously defined settings for the key and secret. The launch URL in our example is example.com:5000. Add a useful name in the "Name" field and keep the other fields empty.
- Click the Submit button to save the app. Note that Canvas does not typically check any of the fields. To test the application, add the bot to an assignment.

And to add the bot to an assignment:

- Go to Courses -> Assignments
- Click + Assignment to add a new assignment and add a new assignment.
- Click the + next to your new assignment to add a sub-assignment.
- Click "More options" to be able to add LTI components.

- Write whatever you want in the screens, an Assignment name is required for the assignment to be saved. Be sure to add "Points" to the assignment if you want to be able to see grades.
- Under "Submission Type" choose "External Tool" Insert example.com:5000 as the URL for the external tool.
- Save the assignment. You will now see an example of the course page you just made, with the OTR bot screen that is provided through LTI.
- Note that the OTR bot only supplies a grade if the assignment is viewed by a student: in the teacher environment, no grade will be passed back to Canvas.

Class documentation

LTI provider that interfaces with the bot. This is a simple setup using Flask, because the PyLTI library already implements Flask decorators.

This module runs a flask server on 0.0.0.0:5000 (which maps to any host, port 5000). It accepts LTI requests using the key and secret in the settings file.

With the playbooks provided, it is also possible to serve these pages via nginx, over port 80.

Adding the bot to your LMS

To add this tool to your LMS, you need to run it on your own server, say example.com. Note that you need to run in production mode, through nginx. This will speed up the server and enable you to use TLS.

In your LMS, enter example.com as the launch URL for this LTI application. The "Consumer key" and "Shared secret" are in the settings file for your environment. In our `settings.development_sample` settings, find *PYLTI_CONFIG*. You can see that we added one consumer under "consumers", which has `__consumer_key__` as his consumer key, and `__lti_secret__` as his secret. It is recommended to change these defaults.

Now it's time to add your bot to the plugins/apps in your LMS. In canvas, follow these steps to add the bot to a course:

- In your course, click Settings and then go to the Apps tab.
- Click Add App
- Choose "Manual entry"
- The only fields that are currently relevant in Canvas are "Consumer key", "Shared secret" and "Launch URL". Follow your previously defined settings for the key and secret. The launch URL in our example is example.com. Add a useful name in the "Name" field and keep the other fields empty.
- Click the Submit button to save the app. Note that Canvas does not typically check any of the fields. To test the application, add the bot to an assignment.

And to add the bot to an assignment:

- Go to Courses -> Assignments
- Click + Assignment to add a new assignment and add a new assignment.
- Click the + next to your new assignment to add a sub-assignment.
- Click "More options" to be able to add LTI components.
- Write whatever you want in the screens, an Assignment name is required for the assignment to be saved. Be sure to add "Points" to the assignment if you want to be able to see grades.
- Under "Submission Type" choose "External Tool" Insert example.com as the URL for the external tool.

- Save the assignment. You will now see an example of the course page you just made, with the OTR bot screen that is provided through LTI.
- Note that the OTR bot only supplies a grade if the assignment is viewed by a student: in the teacher environment, no grade will be passed back to Canvas.

Running LTI in develop mode

By just running `./bot` you should have a werkzeug instance of the LTI bot running. Test it by going to http://localhost:5000/is_up

Running LTI in production

The function `otrbot.__init__.app()` is runnable by gunicorn. You can run a gunicorn instance by simply typing `gunicorn otrbot:app`. Test the setup by going to http://localhost:8000/is_up.

The ansible playbooks should install a systemd service that runs gunicorn and an nginx configuration that serves this through port 443.

The ansible playbooks assume that you ran certbot on your VPS to install certificates. If you haven't, please run this (works on debian):

```
$ sudo apt install certbot
$ certbot standalone
```

otrbot.lti.lti.**app = <Flask 'otrbot.lti.lti'>**
The flask decorator (unfortunately does not follow conventions).

otrbot.lti.lti.**mock_ltid**(*_app=None*, *_request='any'*, *error=None*, *role='any'*, **lti_args*, ***lti_kwargs*)
For debug purposes, replace the normal lti decorator with this one, that returns 'mock-lti' as nickname and does not verify or authenticate

otrbot.lti.lti.**BOT = None**
It is important to initialise the bot variable before calling 'run()'. It should point to an instance of the `OtrBot` class.

otrbot.lti.lti.**get_locale**()
Try to set the locale to something sensible

otrbot.lti.lti.**error_page**(*exception=None*)
Render error page

Parameters **exception** – optional exception

Returns the error.plim template rendered

otrbot.lti.lti.**hello_world**(*lti=<function lti>*)
Indicate the flask app is working. Provided for debugging purposes.

Parameters **lti** – the *lti* object from *pylti*

Returns simple page that indicates the request was processed by the lti provider

otrbot.lti.lti.**index**(**args*, ***kwargs*)
initial access page to the lti provider. This page provides authorization for the user. Contains a button to enter a JID

Parameters **lti** – the *lti* object from *pylti*

Returns index page for lti provider

otrbot.lti.lti.**enter_jid**(**args*, ***kwargs*)
Insert a JID for the bot to add to its roster. The form validates the JID using SleekXMPP's JID constructor. After that, it tries to add the JID to the bot. If that fails (unlikely), the error page is shown.

Parameters **lti** – the *lti* object from *pylti*

Returns Either the form to enter the jid or a redirect to the jid_entered page.

otrbot.lti.lti.**jid_entered**(**args*, ***kwargs*)
A page that confirms that the jid was added to the bot. Contains a button that directs the user tot he "ask_for_secret" page.

Parameters **lti** – the *lti* object from *pylti*

Returns The page confirming the jid was added.

otrbot.lti.lti.**ask_for_secret**(**args*, ***kwargs*)
Show a form to submit a shared secret. When the secret is submitted, it is utf-8 encoded and added to the bot. The bot then checks if the secret is strong enough. If so, it saves it, else, the user is informed about this through Jabber, and asked to enter a new secret through LTI.

If this function is called when a strong secret is already set, the page is skipped and the next page is loaded, where the user is notified that his secret is added.

Parameters **lti** – the *lti* object from *pylti*

Returns Either a page showing the secret form or a page saying that the secret was entered.

otrbot.lti.lti.**enter_bot_secret**(**args*, ***kwargs*)
Form to enter the secret that the bot shares after SMP is successful

otrbot.lti.lti.**run_lti**()
For if you want to run the flask development server directly

OTR bot utility functions

A set of utility functions that are often used throughout the application.

otrbot.core.utils.**fatal**(*code=exit_codes.UNKNOWN_EXCEPTION*, *last_error=None*)
Log a fatal error an exit the application with an exit code.

Parameters

- **code** (*int*) – The exit code the application failed with.
  - exit_codes.INVALID_ARGUMENT
  - exit_codes.BAD_SETTING
  - exit_codes.NO_SETTINGS
  - exit_codes.MISSING_DEPDENDENCY
  - exit_codes.WRONG_TLS_CIPHER_NEGOTIATED = 127
  - exit_codes.UNKNOWN_EXCEPTION
- **last_error** (*str*) – A fatal error message to logger.

otrbot.core.utils.**dummy_i18n**(*msg*)
Dummy i18n function, use when nothing is translated, only defined.

otrbot.core.utils.**generate_secret**(***kwargs*)
Generate a cryptographically secure secret according to a format.

Supply either the format keyword or the length keyword, not both.

In the format string any occurrence of "X" will be replaced by a random character. You can put any separators characters you wish.

Parameters

- **format** (*str*) – Format string, each X will be replaced by a random character ("XXXX-XXXX-XXXX-XXXX").
- **length** (*int*) – Length of the random string.
- **alpha** (*bool*) – Include alphabet in the output (True).
- **case** (*str*) – Case of character options: upper, lower, mixed (upper)
- **numeric** (*bool*) – Include numbers in the output (False).
- **symbols** (*bool*) – Include common symbols in the output (False).
- **options** (*str*) – A string containing all possible characters.

Returns str Secret according to *format*.

otrbot.core.utils.**call_function**(*function*, **args*, ***kwargs*)
calls the function with the appropriate amount of arguments. This enables defining functions with or withouth *args and **kwargs that can both be called through this function.

Parameters

- **function** (*function*) – The function with an arbitrary amount of arguments.
- **args** (*tuple*) – The unnamed arguments of the function (in order).
- **kwargs** (*dict*) – The keyword arguments of the function.

class otrbot.core.utils.**CacheReturn**(*func*)
Cache return of a function/method for a specific set of arguments.

This class memorises all the arguments passed to it and all the return values so it can potentially have a huge memory footprint. Use with caution.

Initialise the CacheReturn class.

Parameters **func** (*function*) – The function object, automatically passed by decorating a function with this class.

otrbot.core.utils.**template_substitute**(*input_string*, ***kwargs*)
shorthand for using string.Template's safe_substitute

otrbot.core.utils.**add_syslog_handler**(*syslog_address='/dev/log'*, *syslog_level='INFO'*)
Add a handler to the logger that logs to syslog.

class otrbot.core.utils.**SleekFilter**(*name=''*)
Filter annoyingly large sleek messages

Initialize a filter.

Initialize with the name of the logger which, together with its children, will have its events allowed through the filter. If no name is specified, allow every event.

AVATAR_MSGS = ['<TYPE>image/png</TYPE><BINVAL>', 'data xmlns="urn:xmpp:avatar:data"']
These strings are typical for messages about avatars:

TRUNCATE = 60
Truncate everything between the first and last TRUNCATE characters

filter(*record*)
Filter a record:

•Truncate messages that contain avatar data

Colourised logging

ANSI colourise the logging stream (works on LINUX/UNIX based systems).

Constants for colours:

attr const BLACK Black

attr const RED Red

attr const GREEN Green

attr const YELLOW Yellow

attr const BLUE Blue

attr const CYAN Cyan

attr const WHITE White

Class documentation

class otrbot.core.colourlog.**ColourFormatter**(**args*, ***kwargs*)
ANSI colourise the logging stream (works on LINUX/UNIX based systems).

Initialise some variables for colourising.

Make cache dict for formats, backup original format string, initialise the parent log formatter.

Parameters

- **args** (*tuple*) – Positional arguments that should be passed to the parent formatter.
- **kwargs** (*dict*) – Keyword arguments that should be passed to the parent formatter. Two keywords are taken from this dict, see below.
- **colours** (*dict*) – A dictionary containing the colour schemes to be used by the logger, see below for more information.
- **no_colour_nl** (*bool*) – Tell the logger not to colour anything after a new line character.

There is a default colour scheme with 2 colour indexes. You can use these colourschemes as follows:

```
formatter = ColourFormatter(
    "$lvl[%(levelname)s]$reset $msg%(name)s %(message)s"
)
```

Note the *$lvl* and *$msg* variables are used as template strings in the format string. Also note there is a *$reset* variable, use this before any *change* in colour, it is automatically added at the end of the string to prevent the terminal from printing coloured strings after the log line was printed.

You can also make custom colour schemes and pass them as a keyword argument (*colours*) when instantiating a ColourFormatter object.

The colours dictionary that can be passed to the *ColourFormatter* is formatted as follows.

```
{
    'lvl': {
        logging.DEBUG: (WHITE, BLUE, False),
        logging.INFO: (BLACK, GREEN, False),
        logging.WARNING: (BLACK, YELLOW, False),
        logging.ERROR: (WHITE, RED, False),
        logging.CRITICAL: (YELLOW, RED, True),
    },
    'msg': {
        logging.DEBUG: (BLUE, None, False),
        logging.INFO: (GREEN, None, False),
        logging.WARNING: (YELLOW, None, False),
        logging.ERROR: (RED, None, False),
        logging.CRITICAL: (RED, None, True),
    }
}
```

The dictionary contains indexes followed by the log levels, followed by a tuple in the form of foreground colour, background color, bold face.

The colourschemes above are the default colours, they colour the *%(levelname)s* in colour scheme *lvl*, which adds background colours as well as foreground colours. The rest of message is can be formatted using the *msg* scheme, which does not add any background colours but does add foreground colours.

The *lvl* and *msg* indexes specify colourschemes. You can make your own indexes, indexes can have arbitrary names but should be formatted be *a-zA-Z0-9-_* and start with *a-zA-Z*. To make use of your colour scheme you need to change your format string. Like this:

```
import logging
logger = logging.getLogger(__name__)
logger.setLevel(level=logging.DEBUG)
handler = logging.StreamHandler()
handler.setFormatter(
    ColourFormatter(
        "$lvl[%(levelname)s]$reset $msg%(name)s %(message)s "
        "$reset"
    )
)
logger.addHandler(handler)

logger.debug("This is detailed information..")
logger.info("This somewhat more useful..")
logger.warn("This might be dangerous..")
logger.error("Something might have gone a bit wrong")
logger.critical("Woah! do something!!")
```

```
\x1b[44;37m[DEBUG]\x1b[0m \x1b[34m__main__ This is detailed information..\x1b[0m
\x1b[42;30m[INFO]\x1b[0m \x1b[32m__main__ This somewhat more useful..\x1b[0m
\x1b[43;30m[WARNING]\x1b[0m \x1b[33m__main__ This might be dangerous..\x1b[0m
\x1b[41;37m[ERROR]\x1b[0m \x1b[31m__main__ Something might have gone a bit wrong\x1b[0m
\x1b[41;33;1m[CRITICAL]\x1b[0m \x1b[31;1m__main__ Woah! do something!!\x1b[0m
```

format(*record*)

Override the normal format method to swap out the format string, then call the parent format method.

Parameters **record** (*object*) – The log record.

get_colour_fmt(*lvl*)

Add a colour to the msg strings and return them.

The result of the Template sting is cached.

Parameters **lvl** (*int*) – The log level.

Settings

Environment variables

If the ENV variable in otrbot.settings.env is set to "production", the otrbot.settings.production will be used. This is the default. You should change this to "development" if you want to work on this project, to prevent your credentials from begin pushed back to Github.

```
ENV = "production"
```

The SSL/TLS configuration options

- TLS is enabled by default but can be switched off for testing, or in case you run your Jabber server on the same physical server.
- SleekXMPP can fall back to SSLv2/v3 if the server doesn't support TLS but don't do this, SSLv3 is already deemed very unsafe.
- You can choose which TLS version the client should use at a minimum.
- You can define ciphers that are allowed, a reasonable list of ciphers that are considered safe at the time of writing (*Apr-'16*) is preconfigured, which should be fine for most modern servers.

otrbot.settings.sample.**ENABLE_TLS = True**
Enable or disable TLS (default: **True — Do not disable!**)

otrbot.settings.sample.**ALLOW_SSL_FALLBACK = False**
Enable or disable SSL (default: **False — Do not enable!**)

otrbot.settings.sample.**TLS_VERSION = 'TLSv1.2'**
TLS version to use (default: **TLSv1**) This can be increased to TLSv1.1 or TLSv1.2 but it requires Python 2.7.9+.

otrbot.settings.sample.**CIPHER_LIST = ['ECDHE-RSA-AES256-GCM-SHA384', 'ECDHE-RSA-AES128-GCM-SHA**
The list of accepted ciphers, this list should be supported by most servers and be reasonably safe at the time of writing (Apr-'16).

ECDHE-RSA-AESXXX-[XXX-]SHA[XXX]

- Elliptic Curve Ephemeral Diffie Hellman key agreement
- Perfect Forward Secret (PFS, because of Ephemeral keys)
- RSA authentication
- AES in CBC or GCM mode for encryption
- SHA1, SHA256 or SHA384 digests

These are safe, and provide excellent performance (GCM is fastest). AES in CBC and GCM mode can be hardware accelerated on most servers. CBC is a block mode cipher, while GCM is a stream cipher, the latter provides better performance for network connections.

ECDHE-ECDSA-AESXXX-[XXX-]SHA[XXX]

- Elliptic Curve Ephemeral Diffie Hellman key agreement
- Perfect Forward Secret (PFS, because of Ephemeral keys)
- ECDSA authentication
- AES in CBC or GCM mode for encryption
- SHA1, SHA256 or SHA384 digests

These are safe, and provide the best performance available, ECDSA requires ECDSA server certificates which are rare at this point in time.

DHE-RSA-AESXXX-[XXX-]SHA[XXX]

- Ephemeral Diffie Hellman key agreement
- Perfect Forward Secret (PFS, because of Ephemeral keys)
- RSA authentication
- AES in CBC or GCM mode for encryption
- SHA1, SHA256 or SHA384 digests

These are safe, but perform significantly worse than ECDHE ciphers.

TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256

- Elliptic Curve Ephemeral Diffie Hellman key agreement
- Perfect Forward Secret (PFS, because of Ephemeral keys)
- RSA authentication
- ChaCha20 stream cipher, provides 256 bits of security
- Poly1305 authenticator to detect forged data.

Safe and lightweight, and provide excellent performance, most notably for mobile devices. AES can be hardware accelerated on most servers, which means it may still outperform ChaCha20-Poly1305. Currently not supported by the most common OpenSSL versions in use.

TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256

- Elliptic Curve Ephemeral Diffie Hellman key agreement
- Perfect Forward Secret (PFS, because of Ephemeral keys)
- ECDSA authentication
- ChaCha20 stream cipher, provides 256 bits of security
- Poly1305 authenticator to detect forged data.

Safe and lightweight, and provide excellent performance, most notably for mobile devices. AES can be hardware accelerated on most servers, which means it may still outperform ChaCha20-Poly1305. ECDSA requires ECDSA server certificates which are rare at this point in time. Currently not supported by the most common OpenSSL versions in use.

TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256

- Ephemeral Diffie Hellman key agreement
- Perfect Forward Secret (PFS, because of Ephemeral keys)
- RSA authentication
- ChaCha20 stream cipher, provides 256 bits of security

- Poly1305 authenticator to detect forged data.

Safe and lightweight, and provide excellent performance, most notably for mobile devices. AES can be hardware accelerated on most servers, which means it may still outperform ChaCha20-Poly1305. Currently not supported by the most common OpenSSL versions in use.

AESXXX-[XXX-]SHA[XXX]

- RSA authentication
- AES in CBC or GCM mode for encryption
- SHA1, SHA256 or SHA384 digests

These good ciphers but they lack Perfect Forward Secrecy, which means if the private keys of the server ever get stolen, all historic data can be trivially decrypted. Performance is very good but at great cost to security. Therefore these are disabled by default. If you experience compatibility issues, enable one by one, top to bottom, until you get a connection. However you should really consider upgrading the server software!

Note: It is still considered too hard to forge a certificate with a SHA1 digest outright but SHA1 is considered deprecated. The ciphers ending in `_SHA` should be disabled for better security. Since some servers do not support SHA256 or higher yet, it is still included for compatibility.

---

otrbot.settings.sample.**CA_CERTS = '/etc/ssl/certs/ca-certificates.crt'**
Location of a PEM file containing all Certificate Authority certificates. This is used for checking the server certificate. To not check the server certificate, set this to *None*.

> **Warning:** Do not disable server certificate checking in production environments, except when running your own XMPP server locally, on the same physical server.

Other settings

otrbot.settings.sample.**ALLOW_SSL_FALLBACK = False**
Enable or disable SSL (default: **False** — **Do not enable!**)

otrbot.settings.sample.**TLS_VERSION = 'TLSv1.2'**
TLS version to use (default: **TLSv1**) This can be increased to `TLSv1.1` or `TLSv1.2` but it requires Python 2.7.9+.

otrbot.settings.sample.**FORMAT_STRINGS = {'bot_title': 'OTR Bot (Canvas)', 'bot_description': 'This is an instance**
The following dict contains the statically defined variables that get formatted into the template strings of messages.

otrbot.settings.sample.**XMPP_ACCOUNTS = {'totem@jabme.eu': {'password': 'quahnahgh6kohTeisixoh5ph', 'avatar_**
A dictionary of the default Jabber account to be used by the bot.

otrbot.settings.sample.**XMPP_DEFAULT_ACCOUNT = 'totem@jabme.eu'**
The JID that the bot uses when the flag -j is not supplied to runclient.

otrbot.settings.sample.**FLASK_CONFIG = {'WTF_CSRF_ENABLED': True, 'PYLTI_URL_FIX': {'https://localhost/'**
The Flask and PyLTI configuration. The values in this dictionary update the standard Flask and PyLTI configuration values.

otrbot.settings.sample.**USE_LTI = 'True'**
When this is true, LTI will be used for authentication. For development purposes, you can turn it off and reach the interface without authentication

otrbot.settings.sample.**LTI_CSS_URL = 'https://hostname.tld/style.css'**
The CSS file that should be added to the head of the LTI (link).

otrbot.settings.sample.**OTR_BOT_MAY_START_SMP = False**
If this is true, the bot is allowed to start SMP by itself as soon as it receives a password. If it's false, the bot will wait for the user to start SMP.

otrbot.settings.sample.**DEFAULT_LOCALE = 'en_GB'**
Default language to use. Take a look at the locales folder to see which locales are available

otrbot.settings.sample.**JID_WHITELIST = {'user@jabber.example.com': {'locale': 'en_GB'}}**
Whitelist for accounts that are authorised to talk to the OTR bot. Can be extended by adding to self.whitelist in bot.py. Users are deleted from the whitelist when their session ends. The whitelist may also contain default values for bot sessions, like the locale

otrbot.settings.sample.**DEFAULT_SECRET = None**
Use this in test-settings to have a default shared secret, so you won't have to bother using an external protocol to insert the secret into the bot.

otrbot.settings.sample.**LOG_USER_MSGS = False**
Log connected users messages to the log file, should be set to False in production to prevent potentially sensitive data from being collected. Note: These are the messages the user sends to the bot.

otrbot.settings.sample.**WARNING_TIMEOUT = 1500**
The number of seconds of inactivity before a user is warned that his session will be killed

otrbot.settings.sample.**SESSION_TIMEOUT = 1800**
The number of seconds of inactivity before a session is killed (should be higher than WARNING_TIMEOUT)

otrbot.settings.sample.**TIMEOUT_CHECK_INTERVAL = 60**
The interval in seconds that the timeouts should be checked on.

otrbot.settings.sample.**SCHEDULED_MSG_SAY = True**
Use a scheduler to delay message with random intervals to make the interaction seem more natural.

otrbot.settings.sample.**THREADED_STATE_SCHEDULER = True**
Thread the state scheduler?

otrbot.settings.sample.**TYPE_SPEED = 20**
How fast should the bot be able to type? (characters per second). This is irrelevant if SCHEDULED_MSG_SAY is set to False.

otrbot.settings.sample.**SYSLOG_ADDRESS = '/dev/log'**
Default syslog address.

otrbot.settings.sample.**SYSLOG_LEVEL = 'INFO'**
Default syslog log level, can be either `DEBUG`, `INFO`, `WARN` or `ERROR`

Exceptions

This page lists all the custom exceptions that are raised by the OTR bot.

Exceptions related to the otrbot client.

exception otrbot.exceptions.client.**JidNotKnown**(*jid=None*)
This exception is raised when a JID is not in the client's _sessions variable when it should be.

Exceptions related to the LTI interface

exception otrbot.exceptions.lti.**BotNotStarted**
Raised if BOT is still None when the server is started. It should contain a reference to the OTR bot.

Exceptions related to the state machine.

exception otrbot.exceptions.statemachine.**StateDoesNotExist**(*state='undefined'*)
Raise this exception when a state is request that does not exist.

exception otrbot.exceptions.statemachine.**CannotHandle**(*event='undefined'*, *state='undefined'*)
Raise this exception when an event that can't be handled is called.

exception otrbot.exceptions.statemachine.**ActionNotAllowed**(*action='undefined'*, *state='undefined'*)
Raise this exception when an action is run that is not allowed by the current state.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

O

A

B

C

D

E

F

G

H

I

J