mushroom Documentation

Release 0.9.20

Michael P. Jung

Jun 11, 2018

Contents

Mushroom is a real-time web messaging framework which is based on gevent and supports WebSockets and long polling transports. It also contains code for inter-process communication via message queues.

CHAPTER 1

Table of contents

1.1 Installation instructions

1.1.1 Prerequisites

The following instructions assume that you are using a Debian or Ubuntu Linux system and have the packages *virtualenvwrapper* and *rabbitmq* installed.

```
$ apt-get install python-dev virtualenvwrapper rabbitmq
```

RabbitMQ is only required if you want to run the the unit tests or you are planning to use the inter process communication using message queues in your own applications. virtualenvwrapper is optional, too. You can also use virtualenv directly or make sure that the required python modules are in your *PYTHONPATH* environment variable.

Create virtualenv and install requirements

```
$ mkvirtualenv --no-site-packages mushroom
$ pip install -r test-requirements.txt
```

Note: Running *setup.py test* without a virtualenv does not work as the selenium tests need to start a second python tasks with the same environment. This task will not be able to find the packages which are installed in the project dir by *setup.py test*.

1.1.2 Running the examples

```
$ workon mushroom
$ cd examples
$ ./chat-server.py
```

All examples start a server at port *39288*. This port was choosen by *random.randint(1024, 65535)*. Once the server is up and running open a browser and open up the URL http://127.0.0.1:39288/.

For more information about the included examples see *Examples*.

1.1.3 Running the unit tests

Prerequisites

Mushroom is compatible with gevent 0.x but the selenium test case does require *gevent.subprocess* which is only part of gevent 1.x. If you plan to run the mushroom unit tests you should use gevent 1.x.

At the time this document was written gevent 1.x was not released and can only be obtained from the Git repository: https://github.com/surfly/gevent

Configure RabbitMQ

In order to run the test suite you need RabbitMQ installed with a vhost */mushroom* and a user with bosh *mushroom* as username and password with full access to the */mushroom* vhost.

Use the following commands to create the vhost, user and permissions:

```
$ rabbitmqctl add_vhost /mushroom
$ rabbitmqctl add_user mushroom mushroom
$ rabbitmqctl set_permissions -p /mushroom mushroom ".*" ".*" ".*"
```

Running the tests

```
$ workon mushroom
$ python setup.py test
```

1.2 Mushroom protocol

1.2.1 General

The mushroom protocol is based on JSON and negotiates the best possible transport with the client. Once the transport negotiation is complete, the protocol becomes transport specific.

1.2.2 Message format

Mushroom uses RPC style messages. All messages are encoded as list with the message type as the first argument. This message format is used for both directions. Thus Mushroom RPC also allows to call methods on the client.

Heartbeat

The heartbeat is used by the client and server to acknowledge messages and keep the connection alive if there has been no traffic for a given amount of time. For transports that do not keep the connection open for an unlimited amount of time this is used for polling.

```
[0, last_message_id]
```

Notification

```
[1, message_id, method_name, data]
```

The method name is required to be a string and SHOULD be namespaced with dots as separator.

Request

This message works uses the same message format as the notification but expects a response.

```
[2, message_id, method_name, data]
```

Response

```
[3, message_id, request_message_id, data]
```

Error response

```
[4, message_id, request_message_id, data]
```

Disconnect

```
[-1]
```

1.2.3 Message id

The message id MUST be implemented as a counter without gaps. It is used to match request and response messages together and provide robust messaging even in case of a unexpected connection timeout.

1.2.4 Transport negotiation

Next generation web applications are probably going to use WebSocket exclusively for transferring realtime data. Until WebSockets are part of all browsers in use, a fallback solution is required to reach a large audience.

Mushroom therefore supports two transports which supports most browsers in use today, while providing the best possible performance for those users living at the bleeding edge.

Currently the following two protocols are supported:

- poll
- websocket

Other transports like JSONP polling, multipart requests and flash sockets are not supported by mushroom as they provide little to no performance benefit over the two supported transports or require a propietary plugin.

Client request

The client sends a POST request to BASE_URL with POST data of the following form:

```
{
    "transports": ["poll", "websocket"]
}
```

Depending on the needs extra fields can be transferred to perform authentication. For example a simple request with an user name and password which only supports long polling as transport could look like:

```
{
    "transports": ["poll"],
    "username": "mobydick",
    "password": "lochness"
}
```

For more sophisticated applications it probably makes sense to handover authentication using some signed credentials:

```
{
    "transports": ["poll", "websocket"],
    "username": "bikeshedder",
    "timestamp": 1341852462,
    "digest": "3369bf6cd89ae1387d2a6b7b0063f7b2f76fb65901dc7bdeea4ac9859e68ed82"
}
```

Note: Those ways of authenticating users is by no means ment to be a defenitive list for authenticating clients. Use whatever authentication method fits your application best. If you are working in a trusted environment or do not need authentication at all feel free to skip it entirely.

Warning: Even though it is possible to use cookies for authentication it is highly discouraged. If a browser falls back to long polling, it will need to transmit the cookies for every single request. This might be fine for very small cookies but still then add bloat that is not required at all as mushroom encodes its own session id into the URL.

Server response

The response from the server will be a JSON of the following format:

```
{
    "transport": "poll",
    "url": "https://example.com/poll/c1108b722b2447f3b603b8ff783233ef/"
}
```

The response for the websocket transport looks simmilar but contains a URL with ws or wss protocol:

```
{
    "transport": "websocket",
    "url": "wss://example.com/websocket/c1108b722b2447f3b603b8ff783233ef/"
}
```

1.2.5 Long polling

All requests to the server must contain a JSON array of messages.

Receiving messages (polling)

Once long polling is decided as transport, the browser is expected to connect to the given URL as soon as possible. If the message array contains a heartbeat, the connection is detected as polling and will not return until there are messages available or the timeout is reached:

```
[
    [0, last_message_id]
]
```

Note: You can also send other messages along with the heartbeat message.

The response is of the following format:

```
[
    message+
]
```

The last message index is a integer and must be given at the next poll request to confirm the receipt of the messages received during the last poll request. This index is used to recover from a timed out poll request without losing messages.

Note: Please note that this requires a disconnecting client to perform one last extra poll request to the server to acknowledge the last received messages before stopping to poll.

Sending messages

The format for sending messages is simple and straightforward:

```
[
    message+
]
```

The message id is a simple counter implemented by the client which is used by the server to filter out duplicate messages. This can be used to filter out already received messages which were retransmitted due to a timeout.

The server response is a simple 200 OK without any data.

Long polling example

Assuming the connection has been up for a while and the server has now reached message id 117. The client has sent 5 messages so far and th next message id is 5 (counting from zero).

1. Poll request (client to server):

```
[
    [0, 117]
]
```

2. Send request (client to server):

```
[
    [1, 5, "player.ready", true],
    [2, 6, "get_time"]
]
```

3. Send response (server to client):

```
(No data, simply a 200 OK)
```

4. Poll response (server to client):

```
[
    [3, 118, 6, 1342106240]
]
```

5. Acknowledge message and continue polling:

```
[
    [1, 118]
]
```

6. ...

1.2.6 WebSocket

WebSockets are bidirectional and have builtin framing. Every message is transferred as a separate frame.

WebSocket example

As in the long polling example the server has reached message id 117 and the last message sent by the client had id 4.

1. Heartbeat (client to server):

```
[0, 117]
```

2. Heartbeat (server to client):

```
[0, 4]
```

3. Notification (client to server):

```
[1, 5, "player.ready", true]
```

4. Request (client to server):

```
[2, 6, "get_time"]
```

5. Response (server to client):

```
[3, 118, 6, 1342106240]
```

1.3 CORS

In order to support Cross-Origin Resource Sharing mushroom utilizes the Access-Control-Allow-Origin header and provides a fallback using a so called CORS iframe. The CORS iframe works by setting the document.domain in the parent frame and inside the iframe at the same time. Once both frames have set their window.domain the parent frame can use the XMLHttpRequest object of the CORS iframe and interact with the server in a natural way.

1.4 Examples

Please see *Installation instructions* how to set up mushroom and run the examples.

The examples can be found in the *examples* directory of the source tree. The following examples are currently available:

1.4.1 Ping example

The browser requests the *ping* method of the server every 2 seconds. When sending the *ping* request -> *ping* is written to the browser window and upon receiving the response a <- *pong* is added.

Server - examples/ping-server.py

```
#!/usr/bin/env python

import example_pythonpath
from example_utils import ExampleServer, ExampleStaticFile


class PingServer(ExampleServer):
    urls = [
        ('/', ExampleStaticFile('ping.html')),
    ] + ExampleServer.urls

    def rpc_ping(self, request):
        return 'pong'


if __name__ == '__main__':
    PingServer.main()
```

Client - examples/ping.html

```
<!DOCTYPE html>

<html>

<head>
```

(continues on next page)

(continued from previous page)

```
        <title>Mushroom Test Client</title>
</head>

<body>

<script type="text/javascript" src="/js/mushroom.js"></script>
<script type="text/javascript">
        var client = new mushroom.Client({
                url: '/'
        });
        client.connect();
        window.setInterval(function() {
                document.write('<div>&rarr; ping</div>');
                client.request('ping', null, function(data) {
                        document.write('<div>&larr; pong</div>');
                });
        }, 2000);
</script>

</body>

</html>
```

1.4.2 Time pusher example

The server pushes the current time every second and the browser displays it.

Server - examples/time-pusher-server.py

```
#!/usr/bin/env python

from time import time

import gevent

import example_pythonpath
from example_utils import ExampleServer, ExampleStaticFile


class TimePusherServer(ExampleServer):
    urls = [
        ('/', ExampleStaticFile('time-pusher.html')),
    ] + ExampleServer.urls

    def server_init(self):
        gevent.spawn(self.time_loop)

    def time_loop(self):
        while True:
            gevent.sleep(1)
            self.sessions.notify('time', time())
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    TimePusherServer.main()
```

Client - examples/time-pusher.html

```
<!DOCTYPE html>

<html>

<head>
        <title>Mushroom Test Client</title>
</head>

<body>

<p id="date"></p>

<script type="text/javascript" src="/js/mushroom.js"></script>
<script type="text/javascript">
        var client = new mushroom.Client({
                url: '/'
        });
        client.connect();
        client.method('time', function(request) {
                var date = new Date(request.data * 1000);
                var dateElement = document.getElementById('date');
                dateElement.innerHTML = date.toString();
        });
</script>

</body>

</html>
```

1.4.3 Chat example

Simple chat room example. The client code uses Knockout and jQuery.

Server - examples/chat-server.py

```
#!/usr/bin/env python

import example_pythonpath
from example_utils import ExampleServer, ExampleStaticFile


class ChatServer(ExampleServer):
    urls = [
        ('/', ExampleStaticFile('chat.html')),
    ] + ExampleServer.urls

    def rpc_message(self, request):
```

(continues on next page)

(continued from previous page)

```
        self.sessions.notify('message', request.data)


if __name__ == '__main__':
    ChatServer.main()
```

Client - examples/chat.html

```
<!DOCTYPE html>

<html>

<head>
        <title>Mushroom Chat Example</title>
</head>
<style type="text/css">
        #messages {
                border: 1px solid #999;
                padding: 0.5em;
                margin: 1em 0;
                height: 20em;
                overflow: auto;
        }
</style>

<body>

<h1>Welcome to Mushroom Chat Example</h1>

<div id="loading" data-bind="visible: !online()">Loading...</div>

<div style="display: none" data-bind="visible: online">

<div data-bind="visible: !usernameSet()">
        <h2>Choose username</h2>
        <form id="login-form" data-bind="submit: setUsername">
                <input name="username" data-bind="value: username, valueUpdate:
↪'afterkeydown'">
                <input type="submit" value="Ok" data-bind="enable: username() !== ''">
        </form>
</div>

<div data-bind="visible: usernameSet">
        <h2>Current discussion</h2>
        <div id="messages">
                <div data-bind="if: messages().length === 0">No chat messages, yet. :-
↪(</div>
                <!-- ko foreach: messages -->
                <div class="message">
                        &lt;<span class="username" data-bind="text: username"></span>&
↪gt;
                        <span class="message" data-bind="text: message"></span>
                </div>
                <!-- /ko -->
        </div>
```

(continues on next page)

(continued from previous page)

```
        <form id="form" data-bind="submit: sendMessage">
                <input name="message" data-bind="value: message, valueUpdate:
↪'afterkeydown'">
                <input type="submit" value="Send" data-bind="enable: message() !== ''
↪">
        </form>
</div>

</div>

<script type="text/javascript" src="/js/mushroom.js"></script>
<script type="text/javascript" src="/js/jquery.js"></script>
<script type="text/javascript" src="/js/knockout.js"></script>
<script type="text/javascript">
$(function() {
        var model = {
                username: ko.observable(''),
                usernameSet: ko.observable(false),
                message: ko.observable(''),
                messages: ko.observableArray(),
                online: ko.observable(false),
                setUsername: function() {
                        this.usernameSet(true);
                },
                sendMessage: function() {
                        client.notify('message', {
                                username: this.username(),
                                message: this.message()
                        });
                        this.message('');
                        return false;
                }
        }
        ko.applyBindings(model);
        var client = new mushroom.Client({
                url: '/'
        });
        client.signals.connected.connect(function() {
                model.online(true);
        });
        client.method('message', function(request) {
                model.messages.push(request.data);
        });
        client.connect();
});
</script>

</body>

</html>
```

1.4.4 Remote control example

Read-eval-print loop which allows remote control of the connected browsers. Start the server and type *help* to get a list of supported commands.

Server - remote-control.py

```
#!/usr/bin/env python

from __future__ import print_function

import cmd
import logging

from gevent import monkey

import example_pythonpath
from example_utils import ExampleServer, ExampleStaticFile


class RemoteControlCmd(cmd.Cmd):

    prompt = 'remote-control> '
    intro = 'Interactive browser remote control\nType "help" for more information.'
    use_rawinput = False

    def __init__(self, server):
        cmd.Cmd.__init__(self)
        self.server = server

    def postcmd(self, stop, line):
        if stop == 'EOF':
            print('^D')
        if stop:
            print('May the maltron be with you!')
            return True

    def do_help(self, args):
        '''Type "help" for the list of available commands and help <command>" for␣
↪details about a specific command.'''
        cmd.Cmd.do_help(self, args)

    def do_exit(self, args):
        '''Exit the console.'''
        return 'exit'

    def do_eval(self, args):
        '''Call eval() on the browser.'''
        self.server.sessions.notify('eval', args)

    def do_print(self, args):
        '''Call print() on the browser.'''
        self.server.sessions.notify('print', args)

    def do_alert(self, args):
        '''Call alert() on the browser.'''
        self.server.sessions.notify('alert', args)

    def do_EOF(self, args):
        '''You can exit the console by typing Ctrl+D.'''
        return 'EOF'
```

(continues on next page)

(continued from previous page)

```
    def do_who(self, args):
        '''Show connected users'''
        if not self.server.sessions:
            print('No sessions connected.')
            return
        print('%d session%s connected:' % (
            len(self.server.sessions),
            's' if len(self.server.sessions) != 1 else ''))
        print('SESSION ID              IP ADDRESS       TRANSPORT')
        print('-------------------------------------------------')
        #####('xxxxxxxxxxxxxxxxxxxxxx  000.000.000.000  ws/poll  ')
        for session in self.server.sessions:
            print('%-22s  %-15s  %-9s' % (session.id,
                session.transport.remote_addr, session.transport.name))

    def do_gauge(self, args):
        '''Set gauge value'''
        self.server.sessions.notify('gauge', args)


class RemoteControlServer(ExampleServer):
    urls = [
        ('/', ExampleStaticFile('remote-control.html')),
    ] + ExampleServer.urls

    def __init__(self, listener):
        super(RemoteControlServer, self).__init__(listener, log=None)


if __name__ == '__main__':
    monkey.patch_sys()
    logging.basicConfig(filename='remote-control.log', level=logging.DEBUG)
    listener = (RemoteControlServer.host, RemoteControlServer.port)
    print('Server running at http://%s:%d/' % listener)
    server = RemoteControlServer(listener)
    server.start()
    rccmd = RemoteControlCmd(server)
    rccmd.cmdloop()
    # XXX how to shutdown cleanly?
```

Client - remote-control.html

```
<!DOCTYPE html>

<html>

<head>
        <title>Remote Control Client</title>
</head>

<body>

<canvas id="gauge"></canvas>

<script type="text/javascript" src="/js/mushroom.js"></script>
```

(continues on next page)

(continued from previous page)

```
<script type="text/javascript" src="/js/gauge.js"></script>
<script type="text/javascript">
        var gauge = new Gauge(document.getElementById('gauge'));
        gauge.maxValue = 100;
        gauge.set(0);

        var client = new mushroom.Client({
                url: '/'
        });
        client.method('eval', function(request) {
                eval(request.data);
        });
        client.method('print', function(request) {
                document.write('<div>' + request.data + '</div>');
        });
        client.method('alert', function(request) {
                alert(request.data);
        });
        client.method('gauge', function(request) {
                var value = parseInt(request.data);
                if (value > gauge.maxValue) {
                        value = gauge.maxValue;
                }
                gauge.set(value);
        });
        client.connect();
</script>

</body>

</html>
```

1.4.5 Webexec example

The server executes an application (currently *ping localhost*) and sends the standard output to all connected browsers.

Server - examples/webexec.py

```
#!/usr/bin/env python

import example_pythonpath
from example_utils import ExampleServer, ExampleStaticFile

import gevent
from gevent import subprocess

class WebexecServer(ExampleServer):
    urls = [
        ('/', ExampleStaticFile('webexec.html')),
    ] + ExampleServer.urls

    def server_init(self):
        gevent.spawn(self.exec_subprocess)
```

(continues on next page)

(continued from previous page)

```
    def exec_subprocess(self):
        proc = subprocess.Popen(['ping', 'localhost'], stdout=subprocess.PIPE)
        while True:
            line = proc.stdout.readline().rstrip()
            if line is None:
                break
            self.sessions.notify('stdout', line)


if __name__ == '__main__':
    WebexecServer.main()
```

Client - examples/webexec.html

```
<!DOCTYPE html>

<html>

<head>
        <title>Mushroom Test Client</title>
</head>

<body>

<p id="output"></p>

<script type="text/javascript" src="/js/mushroom.js"></script>
<script type="text/javascript">
        var client = new mushroom.Client({
                url: '/'
        });
        client.connect();
        client.method('stdout', function(request) {
                var outputElement = document.getElementById('output');
                var lineElement = document.createElement('div');
                lineElement.innerText = request.data;
                outputElement.appendChild(lineElement);
        });
</script>

</body>

</html>
```

1.4.6 Click game example

In this very basic game players must click an appearing square as quick as possible in order to score points.

This example uses jQuery.

Server - examples/click-game.py

```
#!/usr/bin/env python

from time import time
from random import random

import gevent

import example_pythonpath
from example_utils import ExampleServer, ExampleStaticFile


class TimePusherServer(ExampleServer):
    urls = [
        ('/', ExampleStaticFile('click-game.html')),
    ] + ExampleServer.urls

    def server_init(self):
        self.score = 0
        gevent.spawn(self.main_loop)

    def main_loop(self):
        while True:
            gevent.sleep(2)
            x = random()
            y = random()
            self.sessions.notify('target', { 'x': x, 'y': y })

    def rpc_click(self, request):
        self.score += 1
        self.sessions.notify('score', self.score)


if __name__ == '__main__':
    TimePusherServer.main()
```

Client - examples/click-game.html

```
<!DOCTYPE html>

<html>

<head>
        <title>Mushroom Click Game</title>
        <style>
                #score {
                        font-size: 200%;
                        font-weight: bold;
                        margin-bottom: 20px;
                }
                #playing-area {
                        position: relative;
                        width: 440px;
                        height: 440px;
                        background-color: #ccc;
```

(continues on next page)

(continued from previous page)

```
                        border-radius: 20px;
                }
                #target {
                        position: absolute;
                        margin: 20px;
                        height: 40px;
                        width: 40px;
                        background-color: #900;
                        border-radius: 10px;
                        cursor: pointer;
                }
        </style>
</head>

<body>

<div id="score">0</div>

<div id="playing-area">
        <div id="target"></div>
</div>

<script src="/js/mushroom.js"></script>
<script src="/js/jquery.js"></script>
<script>
$(function() {
        var client = new mushroom.Client({
                url: '/'
        });
        var $target = $('#target');
        client.method('target', function(request) {
                $target.css({
                        left: (request.data.x * 400) + 'px',
                        top: (request.data.y * 400) + 'px'
                });
        });
        var $score = $('#score');
        client.method('score', function(request) {
                $score.text(request.data);
        });
        client.signals.connected.connect(function() {
                $('#target').click(function() {
                        client.notify('click');
                });
        });
        client.connect();
});
</script>

</body>

</html>
```

1.5 Python API reference

1.5.1 WSGI application - mushroom.application

class mushroom.application.**Application**(*rpc_handler=None*, *session_handler=None*)

bootstrap(*request*)

request(*request*)

start_session(*request*, *transport*)

1.5.2 HTTP utilities and transports - mushroom.http

Request and response

class mushroom.http.**HttpRequest**(*environ*)

class mushroom.http.**HttpResponse**(*code='200 OK'*, *content=''*, *extra_headers=None*)

class mushroom.http.**JsonResponse**(*data*)

Exceptions

class mushroom.http.**HttpError**(*message=None*)

code = ''

headers = {}

message = ''

class mushroom.http.**HttpUnauthorized**(*auth_scheme=None*)

code = '401 Unauthorized'

class mushroom.http.**HttpNotFound**(*message=None*)

code = '404 Not Found'

class mushroom.http.**HttpMethodNotAllowed**(*allowed_methods*)

code = '405 Method Not Allowed'

class mushroom.http.**HttpInternalServerError**(*message=None*)

code = '500 Internal Server Error'

class mushroom.http.**HttpNotImplemented**(*message=None*)

code = '501 Not Implemented'

Transports

class mushroom.http.**HttpTransport**

get_url(*protocol*, *request*, *session*)

handle_http_request(*request*, *session*)

class mushroom.http.**PollTransport**

get_handshake_data(*request*, *session*)

handle_disconnect(*reconnect=False*)

handle_http_request(*request*, *session*)

name = 'poll'

real_send(*message*)
Perform the actual send operation. This method is only called internally and should not be called from application code. This method is transport specific and must be overwritten.

timeout = 40

class mushroom.http.**WebSocketTransport**

get_handshake_data(*request*, *session*)

handle_disconnect()

handle_http_request(*request*, *session*)

handle_message(*message*)

name = 'ws'

real_send(*message*)
Perform the actual send operation. This method is only called internally and should not be called from application code. This method is transport specific and must be overwritten.

1.5.3 Messaging via message oriented middlewares - mushroom.messaging

class mushroom.messaging.**Client**(*broker_url*, *exchange*, *queue*, *rpc_handler*)

start()

stop()

class mushroom.messaging.**Transport**(*broker_url*, *exchange*, *queue*)

callback(*body*, *message*)

mainloop()

send(*message*, *routing_key=None*)

start()

stop(*join=True*)

1.5.4 Remote procedure calls - mushroom.rpc

Engine

class mushroom.rpc.**Engine**(*transport*, *rpc_handler*)

Transport neutral message factory and mapper between requests and responses. This is the heart of all RPC handling.

handle_message(*message*)

Handle message received from the transport.

Parameters **message** – message to be handled

next_message_id()

Generate the next message id for outbound messages.

Returns: the next message id

notify(*method*, *data=None*, ***kwargs*)

Send a notification.

Parameters

- **method** – name of the method to be called
- **data** – data for the method being called
- **kwargs** – transport specific arguments

request(*method*, *data=None*, *timeout=None*, ***kwargs*)

Send a request and wait for the response or timeout. If no response for the given method is received within *timeout* seconds a *RequestTimeout* exception is raised.

Parameters

- **method** – name of the method to be called
- **data** – data for the method being called
- **timeout** – timeout in seconds for this request
- **kwargs** – transport specific arguments

send(*message*, ***kwargs*)

Hand message over to the transport.

Parameters

- **message** – message to be sent
- **kwargs** – transport specific arguments

RPC handlers

class mushroom.rpc.**MethodDispatcher**(*obj*, *prefix='rpc_'*, *suffix=''*)

Dispatcher implementation that calls methods on an object with a specific prefix and/or suffix. This makes it possible to define objects that provides a set of methods which can then be called by the client.

Note: Using an empty prefix, _ or __ is highly discouraged as it allows the client to call methods like methods like __del__. The same holds true for the suffix. If you really want to dispatch methods without a prefix or suffix it is a good idea to write a custom dispatcher that implements some checks for this.

__call__(*request*)

The *Engine* calls the request handler like it was a function that takes the request as sole argument and returns the response. This function implements the adapter for this interface and makes it possible to use this class as *handler* for the *Engine*.

Parameters **request** – request object

mushroom.rpc.**dummy_rpc_handler**(*request*)

Dummy RPC handler that raises a MethodNotFound exception for all calls. This is useful for applications that do not need do receive any data from the client but only publish data.

Exceptions

exception mushroom.rpc.**RpcError**(*message=''*)

Base class for all exceptions raised from by the *Engine*.

exception mushroom.rpc.**MethodNotFound**(*method_name*)

This error is raised when a method for a *Request* or *Notification* message is not found. This can either happen when a connected client tries to call a server method or the server tries to call a method on the client side.

exception mushroom.rpc.**RequestException**(*data*)

This exception is raised when a *Request* message is answered with an *Error* message.

exception mushroom.rpc.**RequestTimeout**(*message=''*)

This error is raised when a *Request* message is not answered within a specified timeout value. By default the value is set to infinite can be set do a different value when making the request.

Message classes

class mushroom.rpc.**Message**

Base class for all messages.

static from_list(*l*)

Parse a list as defined in the protocol into a message object.

Parameters **l** – list to be parsed

class mushroom.rpc.**Heartbeat**(*last_message_id*)

Heartbeat message

code = 0

static from_list(*l*)
Parse list into a heartbeat message

Parameters **l** – list to be parsed

to_list()
Serialize this message into a list

class mushroom.rpc.**Notification**(*message_id*, *method*, *data=None*)
Notification message

code = 1

static from_list(*l*)
Parse list into a notification message

Parameters **l** – list to be parsed

to_list()
Serialize this message into a list

class mushroom.rpc.**Request**(*message_id*, *method*, *data=None*)
Request message

code = 2

static from_list(*l*)
Parse list into a request message

Parameters **l** – list to be parsed

get_response(*block=True*, *timeout=None*)
Get response for this request.

Parameters

- **block** (*bool*) – block until response is available
- **timeout** (*int or None*) – seconds to wait before raising a *mushroom.rpc.RequestTimeout* error

Return type *mushroom.rpc.Response* or *mushroom.rpc.Error*

Raises *mushroom.rpc.RequestTimeout*

response

to_list()

class mushroom.rpc.**Response**(*message_id*, *request_message_id*, *data=None*)
Response message

code = 3

static for_request(*message_id*, *request*, *data=None*)
Named constructor when the request is known. Some transports need the reference to the original request object when sending the reply for a request. Therefore the *Engine* generates all responses using this method.

Parameters

- **request** – the request which caused this response
- **data** – response data of the method which was called
- **message_id** – id of this message

static from_list(*l*)
Parse list into a response message

Parameters **l** – list to be parsed

to_list()
Serialize this message into a list

class mushroom.rpc.**Error**(*message_id*, *request_message_id*, *data=None*)
Error message

This is the message class and not the exception. The *RpcEngine* will raise a *RequestException* upon receiving this message type.

code = 4

static for_request(*message_id*, *request*, *data=None*)
Named constructor when the request is known. Some transports need the reference to the original request object when sending the reply for a request. Therefore the *RpcEngine* generates all errors using this method.

Parameters

- **request** – the request which caused this response
- **data** – response data of the method which was called
- **message_id** – id of this message

static from_list(*l*)
Parse list into a error message

Parameters **l** – list to be parsed

to_list()
Serialize this message into a list

class mushroom.rpc.**Disconnect**
Disconnect message

code = -1

static from_list(*l*)
Parse list into a disconnect message

Parameters **l** – list to be parsed

to_list()
Serialize this message into a list

1.5.5 Standalone server - mushroom.server

class mushroom.server.**Server**(*listener*, **args*, ***kwargs*)

run()

sessions

1.5.6 Sessions for client-server communication - mushroom.sessions

class mushroom.session.**Session**(*id*, *transport*, *rpc_handler*)

notify(**args*, ***kwargs*)
Send a notification to the connected client.

This method is just a wrapper for the *mushroom.rpc.Engine.request()* method and uses the same arguments.

request(**args*, ***kwargs*)
Send a request to the connected client.

This method is just a wrapper for the *mushroom.rpc.Engine.notify()* method and uses the same arguments.

class mushroom.session.**SessionHandler**

authenticate(*session*, *auth*)

connect(*session*)

disconnect(*session*)

class mushroom.session.**SessionHandlerAdapter**(*obj*, *prefix='session_'*, *suffix=''*)

class mushroom.session.**SessionList**
List of sessions which provides a convenient *notify()* method to notify all sessions. This list also implements copy-on-write (COW) so calls to *add()* and *remove()* are possible during a *notify()* call.

add(*session*)

notify(*method*, *data=None*)

remove(*session*)

mushroom.session.**session_id_generator**()

1.5.7 Simple API - mushroom.simple

class mushroom.simple.**SimpleApplication**(*urls=None*, *rpc_handler=None*, *session_handler=None*)

request(*request*)

class mushroom.simple.**SimpleServer**(*listener*, ***kwargs*)
This is the preferred way of starting to work with mushroom. This server class makes it possible to use mushroom with only a few lines of code by relying on some defaults:

- All RPC methods are prefixed with `rpc_`
- Session handlers calls are prefixed wit `session_`
- The server runs on localhost with port `39288`

In order to use this class create a subclass from it, overwrite the urls attribute and start *YourServer.main()*.

host = '127.0.0.1'

classmethod main()

port = 39288

server_init()

session_authenticate(*session*, *auth*)

session_connect(*session*)

session_disconnect(*session*)

urls = None

class mushroom.simple.**StaticFile**(*filename*)

content_types = {'css': 'text/css', 'gif': 'image/gif', 'html': 'text/html', 'ico':

get(*request*)

load_file()

mushroom.simple.**parse_listener**(*value*)

1.5.8 Transport base classes - mushroom.transport

class mushroom.transport.**UnreliableTransport**

Base class for unreliable transports. Unreliable means that the underlying protocol does not guarantee message delivery by itself. This is typically the case for any protocol that does raw communication without a layer that guarantees message delivery by itself.

connect()

Start transport and connect to the remote side. This method is transport specific and must be overwriten.

disconnect()

Disconnect from the remote side and stop transport. This method is transport specific and must be overwritten.

handle_connect()

handle_disconnect(*reconnect=False*)

handle_heartbeat(*heartbeat*)

handle_message(*message*)

real_send(*message*)

Perform the actual send operation. This method is only called internally and should not be called from application code. This method is transport specific and must be overwritten.

send(*message*)

1.5.9 Utilities - mushroom.utils

class mushroom.utils.**Observable**(*value=None*)

get()

set(*value*)

subscribe(*listener*)

unsubscribe(*listener*)

unsubscribe_all(*listener*)

class mushroom.utils.**Signal**

connect(*handler*)

disconnect(*handler*)

disconnect_all()

send(**args*, ***kwargs*)

1.5.10 Django Support - mushroom.django_support

1.6 FAQ

1.6.1 Which Python versions are supported?

Mushoom supports Python 2 and 3.

1.6.2 Are there plans to support asyncio?

No. While asyncio is great, it depends on the use of `yield from`. This can results in a very unnatural program flow. The goal of mushroom is to make asynchronous I/O as simple and natural as possible. asyncio does not meet this criteria. This is the same reason why mushroom is based on gevent and not Tornado.

1.6.3 Why is there no version 1.0 of mushroom?

Once mushroom is feature complete and provides a stable API version 1.0 will be released. For this to happen all `FIXME`, `TODO` and `XXX` markers need to be gone. The test coverage and documentation need to be way better, too.

1.6.4 Is there Django support?

Mushroom comes with a *django_support* module which provides a `RunserverCommand` base class. This class makes it simple to write custom management commands for starting a mushroom based server.

Max "DebVortex" Brauer has written a django-mushroom package which works in a different way and provides a `runserver_with_mushroom` managment command and `rpc_function` decorators.

1.6.5 Can I get commercial support?

You can also get commercial support from the maintainer and his company Terreon GmbH.

1.7 License

```
Copyright (c) 2012-2013 Michael P. Jung
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
```

(continues on next page)

(continued from previous page)

```
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.

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 HOLDER 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.
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

H

J

L

M

N

O

P

R

S

T

U

W