Bachata Documentation

Release 0.1

Alexey Kinev

October 09, 2016

Contents

Bachata is a chat server toolkit on top of asyncio and Tornado.

CHAPTER 1

Overview

- Requires Python 3.3+
- Requires Tornado running on asyncio event loop
- Implements simple messages queue on Redis LPUSH / BRPOP
- Implements reliable messages delivery on Redis BRPOPLPUSH pattern
- JSON messages format
- Custom messages routing

Reference chapters:

CHAPTER 2

Protocol

Basic messages protocol and design guide.

Messages are presented via JSON strings for sending through network and via standard dict objects for local dispatching.

Messages in Bachata can be of two types:

1. Data messages
2. Transport messages

2.1 Data messages

Data messages are generic messages with any users data. Format is very simple yet flexible, and here's a basic pattern:

```
{
    "id": (str) Message ID,
    "type": (str) Message type,
    "time": (int) Unix-style timestamp in milliseconds,
    "data": (str) / (obj) String or object,
    "from": (str, optional) Sender, may be empty for messages "by system"
    "dest": (str, optional) Destination, may be empty if it's non-adressed
    "sign": (str, optional) Signature
}
```

All fields format except "time" are left for programmers choice!

Here are some ideas on schema design:

1. "id" can be UUID v4, integer number as string or any other unique string
2. "type" field values will be used by routers for filtering, i.e. router for direct messages will pick `"type": "direct"` messages, router for group chats will pick `"type": "group"` messages, etc. **This field must be a string! Integers are reserved for transport messages.**
3. "dest" is a destination identifier, which router should understand, i.e. for direct messages it may be user ID, for group chat messages it may be group chat ID, etc.
4. "data" could be `{"text": "some message text"}` for text messages, `{"location": 33.034:55.0234}` for geographic location messages, may be a mix of many fields `{"text": "hi!", "image": "http://..."}`, etc.
5. "sign" field could be hash of data concatenated with secret token, which is transferred once when user is authenticated, and can be generated and checked both on server and client for preventing unauthorized sending.

There are no really de-facto standard protocols which are simple and at the same time comprehensive, so feel free to design your own!

Maybe later we'll have some best practices section or even a kind of recommended schema, who knows.

2.2 Transport messages

Transport messages are necessary for reliable delivery. Receiver client will notify server on receiving, server will notify senders on deliver, etc.

Transport messages have integer type field!

They have very simple format:

```
{
    "type": (int) Transport type,
    "data": (str) Message ID,
    "sign": (str, optional) Signature
}
```

Types description table:

Type	Description
100	server => sender, server has received message for delivery
200	server <= receiver, receiver has got message
300	server => sender, server has delivered message
1000	server => client, connection "ready" message
1001	"ping" message, should be responded with "pong"
1002	"pong" message, should be sent in response to "ping"

2.3 Base protocol class

class bachata.proto.**BaseProtocol**

Basics for messages loading, dumping and format validation.

dump_message(*message*)

Dump message to str.

load_message(*raw_message*)

Load message to dict from str.

make_message(*id=None*, *type=None*, *time=None*, *dest=None*, *from_=None*, *sign=None*, *data=None*)

Create new message dict.

CHAPTER 3

Messages center

class bachata.**BaseMessagesCenter**(*loop=None*, *proto=None*, *queue=None*)

Messages center provides top-level messages routing.

Parameters

- **loop** – asyncio event loop
- **queue** – Messages queue instance
- **proto** – Messages protocol instance or *BaseProtocol* instance will be used by default

Attributes:

- *proto*: BaseProtocol or subclass instance
- *loop*: asyncio event loop
- *queue*: BaseQueue subclass instance
- *routes*: routes objects list

add_route(*route*)

Add messages route to routing chain, see *BaseRoute*

Message routes are processed in the same order as they added.

Parameters **route** – Route instance

add_socket(*channel*, *websocket*)

Register WebSocket for receiving messages from channel.

Parameters

- **channel** – String channel identifier, based on user id or some hash string
- **websocket** – Tornado WebSocket handler instance

del_route(*route*)

Remove message route, see *BaseRoute*

Parameters **route** – Route instance

del_socket(*channel*, *websocket*)

Unregister WebSocket from receiving messages from channel.

Parameters

- **channel** – String channel identifier, based on user id or some hash string
- **websocket** – Tornado WebSocket handler instance

process(*raw_or_message*, *websocket=None*)
Process message to routing chain and send to WebSockets.

Message is passed to registered routes, they return receivers channels and then message is sent to that channels.

Parameters

- **raw_or_message** – Raw message string or message dict
- **websocket** – WebSocket connection handler received new message, this is optional parameter, because message could also be created at server internally

transport(*message=None*, *websocket=None*)
Process transport layer for messages.

Performs reliable delivery process, notifies sender on successful message delivery.

Parameters

- **message** – Data or transport message to process
- **websocket** – WebSocket which has received message to process

3.1 Routing

class bachata.**BaseRoute**
Base messages route class.

Routes are registered in messages center, every route is responsible for delivering single messages type, i.e. direct users messages, group chat messages, system notifications, etc.

post_process(*message*, *to_channel=None*, *queue=None*)
Post process message after putting it on delivery queue.

Scenarios examples:

•Test if message was delivered after certain timeout and notify via another channel (email, APNS, SMS) if not.

•Send extra service message right after main message.

•etc.

process(*message*, *websocket=None*, *proto=None*)
Process message and return receiver channel for the message.

Scenarios:

•If method returns `None`, message is simply passed to next router in chain.

•If method returns non-empty channel string, then message will be sent to that channel.

•If method returns `True`, then routing chain should be stopped for this message.

Parameters

- **message** – Arbitrary message object.
- **proto** – Messages protocol instance

Returns String channel identifier or `True` or `None`

3.2 Queue

class bachata.**BaseQueue**
Base messages queue class.

add_socket(*channel*, *websocket*, *proto=None*)
Register WebSocket for receiving messages from channel.

Method implementation has to write 'ready' transport message on success or close WebSocket connection.

Parameters

- **channel** – String channel identifier, based on user id or some hash string
- **websocket** – Tornado WebSocket handler instance

check_delivered(*channel*, *message_id*)
Check if message is delivered.

Default implementation always returns True, assuming no messages tracking is implemented initially.

del_socket(*channel*, *websocket*, *proto=None*)
Unregister WebSocket from receiving messages from channel.

Parameters

- **channel** – String channel identifier, based on user id or some hash string
- **websocket** – Tornado WebSocket handler instance
- **proto** – Messages protocol instance

pop_delivered(*channel*, *message_id*, *proto=None*)
Mark message as delivered by ID.

Default implementation is empty.

Parameters

- **channel** – Channel reveived message
- **message_id** – Message ID
- **proto** – Messages protocol instance

put_message(*channels*, *message*, *proto=None*, *from_channel=None*)
Put messages on queue.

Parameters

- **channels** – List of destination channels
- **message** – Message dict object
- **proto** – Messages protocol instance
- **from_channel** – Message from channel

CHAPTER 4

WebSocket handler

class bachata.tornado.**MessagesHandler**(*application*, *request*, ***kwargs*)

Base WebSocket handler for Tornado.

Default implementation works with authentication free channels, it just generates random channel identifier.

Redefine *get_channel()* in subclass if you want to have channels identifiers based on authenticated user.

authenticate()

Authenticate and load current user, asyncio coroutine.

Default implementation is empty, so real authentication logic should be implemented in subclass.

get_channel()

Get channel string identifier for connection. Method must be defined in subclass.

Implementation example:

```
class MyMessagesHandler(talkie.tornado.MessagesHandler):
    def get_channel(self):
        if self.current_user:
            return 'channel:%s' % self.current_user.id
```

get_messages_center()

Get messages center for WebSocket. Method must be defined in subclass.

Implementation example:

```
def get_messages_center(self):
    # bachata.RedisMessagesCenter instance
    return self.application.messages
```

on_auth_error()

Close connection on authorization error.

on_close()

Is called on connection failure.

on_message(*raw_message*)

Process message to messages center.

on_open()

Is called on connection success.

open()

Open WebSocket connection and add socket channel to messages center.

It also performs *authenticate()* coroutine call, so *get_channel()* can rely on current authenticated user.

CHAPTER 5

Redis stack

class bachata.redis.**RedisMessagesCenter**(*loop=None*, *conn_params=None*, *reliable=False*)
Messages center on top of Redis LPUSH / BRPOP pattern.

After creating messages center instance *init()* coroutine also must be called.

Parameters

- **loop** – asyncio event loop
- **conn_params** – Redis connection params as dict
- **reliable** – Use reliable queue or simple queue, default is False

init()
Setup main Redis connection for queue.

class bachata.redis.**RedisQueue**(*loop=None*, *conn_params=None*)
Messages queue on top of Redis LPUSH / BRPOP pattern.

Schema description:

1. Messages are LPUSH'ed to list with "{channel}" key.
2. Receiver just listens for "{channel}" list updates with BRPOP.

Parameters

- **loop** – asyncio event loop
- **websocket** – WebSocket handler instance
- **conn_params** – Redis connection params

add_socket(*channel*, *websocket*, *proto=None*)
Register WebSocket for receiving messages from channel.

Parameters

- **channel** – String channel identifier, based on user id or some hash string
- **websocket** – Tornado WebSocket handler instance
- **proto** – Messages protocol instance

connect()
Setup main Redis connection.

del_socket(*channel*, *websocket*, *proto=None*)
Unregister WebSocket from receiving messages from channel.

Parameters

- **channel** – String channel identifier, based on user id or some hash string
- **websocket** – Tornado WebSocket handler instance
- **proto** – Messages protocol instance

listen_queue(*channel*, *websocket*)
Start queue listener for channel and WebSocket connection.

put_message(*channels*, *message*, *proto=None*, *from_channel=None*)
Put messages on queue.

Parameters

- **channels** – List of destination channels
- **message** – Message dict object
- **proto** – Messages protocol instance
- **from_channel** – Message from channel

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

b