pyryver

Release 0.2.0

May 12, 2020

Contents

CHAPTER 1

Introduction

1.1 Prerequisites

`pyryver` requires Python 3.6 or later, and is regularly tested against Python 3.6 & Python 3.8. Our only dependency is on aiohttp.

You may also wish to read aiohttp's information about optional prerequisites for high-performance workloads.

1.2 Installation

Installing `pyryver` can either be accomplished by cloning our git repository and doing the normal `setup.py install`, or using PyPI:

```
# normal
pip install -U pyryver
# if you have multiple versions of python
python3 -m pip install -U pyryver
# if you use windows
py -3 -m pip install -U pyryver
```

1.3 Key Information

In Ryver's API, the base class is a Chat. This, although somewhat unintuitive, does make sense: all of Ryver's functionality can be accessed through one of many interfaces, all of which support chatting. As such, `pyryver`'s API and this documentation often uses the word "chat" to refer to "users, teams and forums". We also use the term "group chat" to refer to both teams and forums, and you might see them referred to as "conferences" within the code since that's what Ryver appears to call them (especially within the WebSocket API).

We also use the term "logged-in" user to refer to whichever user who's credentials were passed when creating the Ryver session.

1.4 Quickstart

The core of the `pyryver` API is the *`pyryver.ryver.Ryver`* object, which represents a session with the Ryver OData HTTP API.

```
async with pyryver.Ryver("organization_url", "username", "password") as ryver:
    pass
```

The `Ryver` object also stores (and can cache) some information about the Ryver organization, specifically lists of all chats.

These can be loaded either with the type-specific *`pyryver.ryver.Ryver.load_users`*, *`pyryver.ryver.Ryver.load_teams`* and *`pyryver.ryver.Ryver.load_forums`* or with *`pyryver.ryver.Ryver.load_chats`*. There's also `pyryer.ryver.Ryver.load_missing_chats` which won't update already loaded chats, which can be useful.

```
async with pyryver.Ryver("organization_url", "username", "password") as ryver:
    await ryver.load_chats()

    a_user = ryver.get_user(username="tylertian123")
    a_forum = ryver.get_groupchat(display_name="Off-Topic")
```

Notice that since we grab *all* the chats once at the beginning, the specific chat lookup methods do not need to be awaited, since they just search within pre-fetched data. Also notice that searching for users and group chats are in separate methods; either a *`pyryver.objects.Forum`* or *`pyryver.objects.Team`* is returned depending on what gets found.

Most of the functionality of `pyryver` exists within these chats, such as sending/checking messages and managing topics. Additional, more specific methods (such as user and chat membership management) can also be found within the different *`pyryver.objects.Chat`* subclasses. For example, the following code will scan the most recent 50 messages the logged-in user sent to `tylertian123` and inform them of how many times an ellipsis occurred within them.

```
async with pyryver.Ryver("organization_url", "username", "password") as ryver:
    await ryver.load_chats()

    a_user = ryver.get_user(username="tylertian123")
    # a_forum = ryver.get_groupchat(display_name="Off-Topic")

    tally = 0
    for message in await a_user.get_messages(50):
        if "..." in message.get_body():
            tally += 1

    await a_user.send_message("There's been an ellipsis in here {} times".
↪format(tally))
```

For more information on how to use *`Chats`* and other Ryver data types, use the *Ryver entities reference*.

1.5 Realtime Quickstart

Building on the previous example, what if we want our terrible ellipsis counting bot to give live updates? We can use the **realtime** API! The realtime interface is centred around the *`pyryver.ryver_ws.RyverWS`* object, which can be obtained with `Ryver.get_live_session()`. Unlike the rest of the API, the realtime API is largely event driven. For example:

```
async with pyryver.Ryver("organization_url", "username", "password") as ryver:
    await ryver.load_chats()

    a_user = ryver.get_user(username="tylertian123")

    async with ryver.get_live_session() as session:
        @session.on_chat
        async def on_chat(msg):
            pass

        await session.run_forever()
```

There are a few things to notice here: firstly, that we can set event handlers with the various `on_` decorators of the *pyryver.ryver_ws.RyverWS* instance (you could also call these directly like any other decorator if you want to declare these callbacks without having obtained the *pyryver.ryver_ws.RyverWS* instance yet), and secondly that the realtime API starts as soon as it is created. *pyryver.ryver_ws.RyverWS.run_forever()* is a helper that will run until something calls *pyryver.ryver_ws.RyverWS.close()*, which can be called from within event callbacks safely.

The contents of the `msg` parameter passed to our callback is currently just the raw JSON message from the Ryver WebSocket system. (documentation as to the fields present are available TODO) In the `chat` message, there are two fields our "bot" needs to care about: "to", which specifies which chat the message was posted in, and "text", which is the content of the message. "from" refers to the message's creator. Perhaps unintuitively, the "to" field should be referring to our user's chat, since we're looking at a private DM. For group chats, you'd expect the chat's ID here.

In fact, you would expect the chat's **JID** here, since the websocket system uses JIDs to refer to chats. Using this information, we can complete our terrible little bot:

Note: The reason for the separate IDs is because the "ratatoskr" chat system appears to be built on XMPP, which uses these "JabberID"s to refer to users and groups.

```
async with pyryver.Ryver("organization_url", "username", "password") as ryver:
    await ryver.load_chats()

    a_user = ryver.get_user(username="tylertian123")
    me = ryver.get_user(username="username")

    async with ryver.get_live_session() as session:
        @session.on_chat
        async def on_chat(msg):
            # did the message come from a_user and was sent via DM to us?
            if msg["to"] == me.get_jid() and msg["from"] == a_user.get_jid():
                # did the message contain "..."?
                if "..." in msg["text"]:
                    # send a reply via the non-realtime system (slow)
                    # await a_user.send_message("Hey, that ellipsis is _mean_!")
                    # send a reply via the realtime system
                    await session.send_chat(a_user, "Hey, that ellipsis is _mean_!")

        await session.run_forever()
```

It's important to note here that although the non-realtime API is perfectly accessible (and sometimes necessary) to use in event callbacks, it's often faster to use corresponding methods in the *pyryver.ryver_ws.RyverWS* instance whenever possible. For some ephemeral actions like typing indicators and presence statuses, the realtime API is the *only* way to accomplish certain tasks.

For more information on how to use the realtime interface, use the *live session reference*.

CHAPTER 2

API Reference

This is the full reference of everything in pyryver.

Note: In all cases where a fully qualified name to something is used, such as `pyryver.ryver.Ryver`, any submodule can be ignored, as they are all imported into the global `pyryver` scope.

2.1 Session

class `pyryver.ryver.`**`Ryver`**(*org: str = None, user: str = None, password: str = None, cache: Type[pyryver.cache_storage.AbstractCacheStorage] = None*)

A Ryver session contains login credentials and organization information.

This is the starting point for any application using pyryver.

If the organization or username is not provided, it will be prompted using input(). If the password is not provided, it will be prompted using getpass().

The cache is used to load the chats data. If not provided, no caching will occur.

If a valid cache is provided, the chats data will be loaded in the constructor. Otherwise, it must be loaded through load_forums(), load_teams() and load_users() or load_chats().

Parameters

- **`org`** – Your organization's name. (as seen in the URL)
- **`user`** – The username to authenticate with.
- **`password`** – The password to authenticate with.
- **`cache`** – The aforementioned cache.

`get_chat`(***kwargs*) → pyryver.objects.Chat

Get a specific forum/team/user.

If no query parameters are supplied, more than one query parameters are supplied or forums/teams/users are not loaded, raises `ValueError`.

Allowed query parameters are:

- id
- jid

Returns none if not found.

`async with get_live_session`() → pyryver.ryver_ws.RyverWS
Get a live session.

The session is not started unless start() is called or if it is used as a context manager.

`get_user`(***kwargs*) → pyryver.objects.User
Get a specific user.

If no query parameters are supplied, more than one query parameters are supplied or users are not loaded, raises `ValueError`.

Allowed query parameters are:

- id
- jid
- username
- display_name
- email

Returns none if not found.

`get_groupchat`(***kwargs*) → pyryver.objects.GroupChat
Get a specific forum/team.

If no query parameters are supplied, more than one query parameters are supplied or forums/teams are not loaded, raises `ValueError`.

Allowed query parameters are:

- id
- jid
- name
- nickname

Returns none if not found.

`await get_object`(*obj_type: str*, *obj_id: int*) → pyryver.objects.Object
Get an object from Ryver with a type and ID.

This method sends requests.

Parameters

- **`obj_type`** – The type of the object to retrieve, a constant beginning with `TYPE_` in *pyryver.util*.
- **`obj_id`** – The object's ID.

`await get_info`() → Dict[str, Any]

Get organization and user info.

This method returns an assortment of info. It is currently the only way to get avatar URLs for users/teams/forums etc. The results (returned mostly verbatim from the Ryver API) include:

- Basic user info - contains avatar URLs ("me")
- User UI preferences ("prefs")
- Ryver app info ("app")
- Basic info about all users - contains avatar URLs ("users")
- Basic info about all teams - contains avatar URLs ("teams")
- Basic info about all forums - contains avatar URLs ("forums")
- All available commands ("commands")
- "messages" and "prefixes", the purpose of which are currently unknown.

This method sends requests.

`async for notification in get_notifs`(*unread: bool = False*, *top: int = -1*, *skip: int = 0*) → AsyncIterator[pyryver.objects.Notification]

Get all the user's notifications.

This method sends requests.

Parameters

- **`unread`** – If True, only return unread notifications.
- **`top`** – Maximum number of results.
- **`skip`** – Skip this many results.

`await mark_all_notifs_read`() → int

Marks all the user's notifications as read.

This method sends requests.

Returns how many notifications were marked as read.

`await mark_all_notifs_seen`() → int

Marks all the user's notifications as seen.

This method sends requests.

Returns how many notifications were marked as seen.

`await upload_file`(*filename: str*, *filedata: Any*, *filetype: str = None*) → pyryver.objects.Storage

Upload a file to Ryver.

Although this method uploads a file, the returned object is an instance of Storage. Use Storage.get_file() to obtain the actual File object.

Parameters

- **`filename`** – The filename to send to Ryver. (this will show up in the UI if attached as an embed, for example)
- **`filedata`** – The file's raw data, sent directly to `aiohttp.FormData.add_field()`.

await load_chats() → None
Load the data of all users/teams/forums.

This refreshes the cached data if a cache is supplied.

This method sends requests.

await load_missing_chats() → None
Load the data of all users/teams/forums if it does not exist.

Unlike load_chats(), this does not update the cache.

This method could send requests.

await load_users() → None
Load the data of all users.

This refreshes the cached data if a cache is supplied.

This method sends requests.

await load_forums() → None
Load the data of all forums.

This refreshes the cached data if a cache is supplied.

This method sends requests.

await load_teams() → None
Load the data of all teams.

This refreshes the cached data if a cache is supplied.

This method sends requests.

await close()
Close this session.

2.2 Realtime Client

class pyryver.ryver_ws.**RyverWS**(*ryver: Ryver*)
A live Ryver session using websockets.

You can construct this manually, although it is recommended to use `Ryver.get_live_session()`.

@**on_chat**(*func*)
The on chat message coroutine decorator.

This coroutine will be started as a task when a new chat message arrives. It should take a single argument, the chat message data.

@**on_chat_deleted**(*func*)
The on chat message deleted coroutine decorator.

This coroutine will be started as a task when a chat message is deleted. It should take a single argument, the chat message data.

@**on_chat_updated**(*func*)
The on chat message updated coroutine decorator.

This coroutine will be started as a task when a chat message is updated. It should take a single argument, the chat message data.

@on_connection_loss (*func*)

The on connection loss coroutine decorator.

This coroutine will be started as a task when the connection is lost.

EVENT_REACTION_ADDED = '/api/reaction/added'

EVENT_REACTION_REMOVED = '/api/reaction/removed'

EVENT_ALL = ''

@on_event (*event_type: str*)

The on event coroutine decorator for a specific event or all unhandled events.

This coroutine will be started as a task when a new event arrives with the specified type. If the event_type is None or an empty string, it will be called for all events that are unhandled. It should take a single argument, the event data.

Parameters **event_type** – The event type to listen to, one of the constants in this class starting with `EVENT_` or *RyverWS.EVENT_ALL* to receieve all otherwise unhandled messages.

MSG_TYPE_ALL = ''

@on_msg_type (*msg_type*)

The on message type coroutine decorator for a specific message type or all unhandled messages.

This coroutine will be started as a task when a new message arrives with the specified type. If the msg_type is None or an empty string, it will be called for all messages that are unhandled. It should take a single argument, the message data.

Parameters **msg_type** – The message type to listen to, one of the constants in this class starting with `MSG_TYPE_` or *RyverWS.MSG_TYPE_ALL* to receieve all otherwise unhandled messages.

await send_chat (*to_chat: pyryver.objects.Chat*, *msg: str*)

Send a chat message to a chat.

Parameters

- **to_chat** – The chat to send the message to.
- **msg** – The message contents.

async with typing (*to_chat: pyryver.objects.Chat*) → pyryver.ryver_ws.RyverWSTyping

Get a context manager that keeps sending a typing indicator to a chat.

Useful for wrapping long running operations, like:

```
async with session.typing(chat):
    print("do something silly")
    await asyncio.sleep(4)
    await session.send_chat(chat, "done") # or do it outside the with, doesn
↪'t matter
```

Parameters **to_chat** – Where to send the typing status.

await send_typing (*to_chat: pyryver.objects.Chat*)

Send a typing indicator to a chat identified by JID.

The typing indicator automatically clears after a few seconds or when a message is sent.

Parameters **to_chat** – Where to send the typing status.

PRESENCE_AVAILABLE = 'available'

PRESENCE_AWAY = 'away'

PRESENCE_DO_NOT_DISTURB = 'dnd'

PRESENCE_OFFLINE = 'unavailable'

await send_presence_change(*presence: str*)
Send a presence change message.

Parameters **presence** – The new presence, a constant in this class starting with `PRESENCE_`

Note: If you use this class as an `async with` context manager, you don't need to call these two methods, unless you want to break out of a *`RyverWS.run_forever()`*.

---

await start()
Start the session.

await close()
Close the session.

Any future operation after closing will result in a ClosedError being raised.

await run_forever()
Run forever, or until the connection is closed.

class pyryver.ryver_ws.**RyverWSTyping**(*rws: pyryver.ryver_ws.RyverWS*, *to: pyryver.objects.Chat*)
A context manager returned by *`RyverWS`* to keep sending a typing indicator.

You should not create this class yourself, rather use `RyverWS.start_typing()` instead.

start()
Start sending the typing indicator.

await stop()
Stop sending the typing indicator.

Note that the typing indicator doesn't clear immediately. It will clear by itself after about 3 seconds, or when a message is sent.

class pyryver.ryver_ws.**ClosedError**
Bases: `Exception`

An exception raised to indicate that the session has been closed.

2.3 Data Models

class pyryver.objects.**Object**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Base class for all Ryver objects.

Parameters

- **ryver** – The parent `pyryver.pyryver.Ryver` instance.
- **obj_type** – The object's type, a constant beginning with `TYPE_` in *pyryver.util*.

get_entity_type() → str
Get the entity type of this object.

get_id() → Any
Get the ID of this object.

This is usually an integer, however for messages it is a string instead.

get_raw_data() → dict
Get the raw data of this object.

The raw data is a dictionary directly obtained from parsing the JSON response.

get_ryver() → pyryver.ryver.Ryver
Get the Ryver session this object was retrieved from.

get_type() → str
Get the type of this object.

2.3.1 Chats

class pyryver.objects.**Chat**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Object*

Any Ryver chat you can send messages to.

E.g. Teams, forums, user DMs, etc.

await create_topic(*subject: str*, *body: str*, *creator: pyryver.objects.Creator = None*) → pyryver.objects.Topic
Create a topic in this chat.

This method sends requests.

Returns the topic created.

Parameters

- **subject** – The subject (or title) of the new topic.
- **body** – The contents of the new topic.
- **creator** – The overriden creator; optional, if unset uses the logged-in user's profile.

get_jid() → str
Get the JID (JabberID) of this chat.

The JID is used in the websockets interface.

await get_message_from_id(*id: str*, *before: int = 0*, *after: int = 0*) → List[pyryver.objects.Message]
Get a message from an ID, optionally also messages before and after it too.

> **Warning:** Before and after cannot exceed 25 messages, otherwise an HTTPError will be raised with the error code 400 Bad Request.

This method sends requests.

This method does not support top/skip.

Parameters

- **id** – The ID of the message to retrieve, and the reference point for the `before` and `after` parameters.
- **before** – How many extra messages to retrieve before the specified one.

- **after** – How many extra messages to retrieve after the specified one.

await get_messages(*count: int*) → List[pyryver.objects.ChatMessage]

Get a number of messages (most recent first) in this chat.

This method sends requests.

Parameters **count** – Maximum number of results.

abstractmethod get_name() → str

Get the name of this chat.

async for ... in get_topics(*archived: bool = False*, *top: int = -1*, *skip: int = 0*) → AsyncIterator[pyryver.objects.Topic]

Get all the topics in this chat.

This method sends requests.

Parameters

- **archived** – If True, only include archived topics in the results, otherwise, only include non-archived topics.
- **top** – Maximum number of results; optional, if unspecified return all results.
- **skip** – Skip this many results.

await send_message(*message: str*, *creator: pyryver.objects.Creator = None*) → str

Send a message to this chat.

Specify a creator to override the username and profile of the message creator.

This method sends requests.

Returns the ID of the chat message sent. Note that message IDs are strings.

Parameters

- **message** – The message contents.
- **creator** – The overriden creator; optional, if unset uses the logged-in user's profile.

class pyryver.objects.**GroupChat**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)

Bases: *pyryver.objects.Chat*

A Ryver team or forum.

await get_member(*id: int*) → pyryver.objects.GroupChatMember

Get a member by user ID.

This method sends requests.

If the user is not found, this method will return None.

async for ... in get_members(*top: int = -1*, *skip: int = 0*) → AsyncIterator[pyryver.objects.GroupChatMember]

Get all the members of this chat.

This method sends requests.

Parameters

- **top** – Maximum number of results; optional, if unspecified return all results.
- **skip** – Skip this many results.

get_name() → str

Get the name of this chat.

get_nickname() → str
Get the nickname of this chat.

class pyryver.objects.**Forum**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.GroupChat*

A Ryver forum.

class pyryver.objects.**Team**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.GroupChat*

A Ryver team.

2.3.2 User

class pyryver.objects.**User**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Chat*

A Ryver user.

ROLE_ADMIN = 'ROLE_ADMIN'

ROLE_GUEST = 'ROLE_GUEST'

ROLE_USER = 'ROLE_USER'

await create_topic(*from_user: pyryver.objects.User*, *subject: str*, *body: str*, *creator: pyryver.objects.Creator = None*) → pyryver.objects.Topic
Create a topic in this user's DMs.

This method sends requests.

Returns the topic created.

Parameters

- **from_user** – The user that will create the topic; must be the same as the logged-in user.
- **subject** – The subject (or title) of the created topic.
- **body** – The contents of the created topic.

get_about() → str
Get this user's About.

get_activated() → bool
Get whether this user's account is activated.

get_display_name() → str
Get the display name of this user.

get_email_address() → str
Get this user's Email Address.

get_name() → str
Get the display name of this user.

get_role() → str
Get this user's role in their profile.

> **Note:** This is different from *get_roles()*. While this one gets the "Role" of the user from the profile, get_roles() gets the user's roles in the organization (user, guest, admin).

get_roles() → List[str]
Get this user's role in the organization.

> **Note:** This is different from *get_role()*. While this one gets the user's roles in the organization (user, guest, admin), `get_role()` gets the user's role from their profile.

get_time_zone() → str
Get this user's Time Zone.

get_username() → str
Get the username of this user.

is_admin() → bool
Get whether this user is an org admin.

await set_activated(*activated: bool*) → None
Activate or deactivate the user. Requires admin.

This method sends requests.

> **Note:** This also updates these properties in this object.

await set_org_role(*role: str*) → None
Set a user's role in this organization, as described in *get_roles()*.

This can be either ROLE_USER, ROLE_ADMIN or ROLE_GUEST.

This method sends requests.

> **Note:** This also updates these properties in this object.

await set_profile(*display_name: str = None*, *role: str = None*, *about: str = None*) → None
Update this user's profile.

If any of the arguments are None, they will not be changed.

This method sends requests.

> **Note:** This also updates these properties in this object.

Parameters

- **display_name** – The user's new display_name.
- **role** – The user's new role, as described in *get_role()*.
- **about** – The user's new "about me" blurb.

2.3.3 Group Chat Member

class pyryver.objects.**GroupChatMember**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Object*

A member in a forum or team.

ROLE_ADMIN = 'ROLE_TEAM_ADMIN'

ROLE_MEMBER = 'ROLE_TEAM_MEMBER'

get_role() → str
Get the role of this member.

get_user() → pyryver.objects.User
Get this member as a *User* object.

is_admin() → bool
Get whether this member is an admin of their forum.

> **Warning:** This method does not check for org admins.

2.3.4 Messages

class pyryver.objects.**Message**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Object*

Any generic Ryver message, with an author, body, and reactions.

get_attached_file() → pyryver.objects.File
Get the file attached to this message, if there is one.

Note that files obtained from this only have a limited amount of information, including the ID, name, URL, size and type. Attempting to get any other info will result in a KeyError. To obtain the full file info, use `Ryver.get_object()` with *TYPE_FILE* and the ID.

Returns None otherwise.

await get_author() → pyryver.objects.User
Get the author of this message, as a *User* object.

This method sends requests.

abstractmethod get_author_id() → int
Get the ID of the author of this message.

abstractmethod get_body() → str
Get the body of this message.

get_creator() → pyryver.objects.Creator
Get the Creator of this message.

Note that this is different from the author. Creators are used to override the display name and avatar of a user. If the username and avatar were not overridden, this will return None.

get_reaction_counts() → dict
Count the number of reactions for each emoji on a message.

Returns a dict of {emoji: number_of_reacts}.

get_reactions() → dict
Get the reactions on this message.

Returns a dict of {emoji: [users]}.

await react(*emoji: str*) → None
React to a message with an emoji.

This method sends requests.

Parameters **emoji** – The string name of the reacji (e.g. "thumbsup").

class pyryver.objects.**ChatMessage**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)

Bases: *pyryver.objects.Message*

A Ryver chat message.

await delete() → None

Deletes the message.

await edit(*body: str*, *creator: pyryver.objects.Creator = None*) → None

Edit the message.

Parameters

- **body** – The new message content.
- **creator** – The new message creator; optional, if unset left as-is.

get_author_id() → int

Get the ID of the author of this message.

get_body() → str

Get the message body.

await get_chat() → pyryver.objects.Chat

Get the chat that this message was sent to, as a *Chat* object.

This method sends requests.

get_chat_id() → int

Get the id of the chat that this message was sent to, as an integer.

Note that this is different from *get_chat()* as the id is stored in the message data and is good for most API purposes while get_chat() returns an entire Chat object, which might not be necessary depending on what you're trying to do.

get_chat_type() → str

Gets the type of chat that this message was sent to, as a string.

This string will be one of the ENTITY_TYPES values

await react(*emoji: str*) → None

React to a message with an emoji.

This method sends requests.

Parameters **emoji** – The string name of the reacji (e.g. "thumbsup").

class pyryver.objects.**Topic**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)

Bases: *pyryver.objects.Message*

A Ryver topic in a chat.

get_author_id() → int

Get the ID of the author of this topic.

get_body() → str

Get the body of this topic.

async for ... in get_replies(*top: int = -1*, *skip: int = 0*) → AsyncIterator[pyryver.objects.TopicReply]

Get all the replies to this topic.

This method sends requests.

Parameters

- **top** – Maximum number of results; optional, if unspecified return all results.
- **skip** – Skip this many results.

get_subject() → str
Get the subject of this topic.

await reply(*message: str*, *creator: pyryver.objects.Creator* = *None*) → pyryver.objects.TopicReply
Reply to the topic.

This method sends requests.

For unknown reasons, overriding the creator does not work for this method.

Parameters **message** – The reply content

class pyryver.objects.**TopicReply**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Message*

A reply on a topic.

get_author() → pyryver.objects.User
Get the author of this reply, as a *User* object.

Unlike the other implementations, this does not send any requests.

get_author_id() → int
Get the ID of the author of this reply.

get_body() → str
Get the body of this message.

get_topic() → pyryver.objects.Topic
Get the topic this reply was sent to.

2.3.5 Files

class pyryver.objects.**Storage**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Object*

Generic storage, e.g. uploaded files.

Note that while storage objects contain files, the File class does not inherit from this class.

get_file() → pyryver.objects.File
Get the file stored.

class pyryver.objects.**File**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Object*

An uploaded file.

This class also contains constants for some common MIME types.

await delete() → None
Delete this file.

This method sends requests.

await download_data() → bytes
Download the file data.

This method sends requests.

get_MIME_type() → str
Get the MIME type of this file.

get_name() → str
Get the name of this file.

get_size() → int
Get the size of this file in bytes.

get_title() → str
Get the title of this file.

get_url() → str
Get the URL of this file.

request_data() → aiohttp.client_reqrep.ClientResponse
Get the file data.

Returns an aiohttp request response to the file URL.

2.3.6 Notification

class pyryver.objects.**Notification**(*ryver: pyryver.ryver.Ryver*, *obj_type: str*, *data: dict*)
Bases: *pyryver.objects.Object*

A Ryver user notification.

PREDICATE_COMMENT = 'commented_on'

PREDICATE_GROUP_MENTION = 'group_mention'

PREDICATE_MENTION = 'chat_mention'

PREDICATE_TASK_COMPLETED = 'completed'

get_new() → bool
Get whether this notification is new.

get_object() → dict
Get the "object" of this notification.

The exact nature of this field is not yet known, but it seems to be the target of an @mention for mentions, the topic for topic comments, or the task for task activities.

get_object_entity_type() → str
Get entity type of the "object" of this notification.

The exact nature of this field is not yet known, but it seems to be the target of an @mention for mentions, the topic for topic comments, or the task for task activities.

get_object_id() → int
Get the ID of the "object" of this notification.

The exact nature of this field is not yet known, but it seems to be the target of an @mention for mentions, the topic for topic comments, or the task for task activities.

get_predicate() → str
Get the "predicate", or type, of this notification.

E.g.

- chat_mention - User was @mentioned
- group_mention - User was mentioned through @team or @here

- commented_on - A topic was commented on
- completed - A task was completed

get_subject_entity_type() → str
Get the entity type of the "subject" of this notification.

The exact nature of this field is not yet known, but it seems to be the user that did the action which caused this notification.

get_subject_id() → int
Get the ID of the "subject" of this notification.

The exact nature of this field is not yet known, but it seems to be the user that did the action which caused this notification.

get_subjects() → List[dict]
Get the "subjects" of this notification.

The exact nature of this field is not yet known, but it seems to be the user that did the action which caused this notification. It is also unknown why this is an array, as it seems to only ever contain one element.

get_unread() → bool
Get whether this notification is unread.

get_via() → dict
Get the "via" of this notification.

The exact nature of this field is not yet known, but it seems to contain information about whatever caused this notification. For example, the chat message of an @mention, the topic reply for a reply, etc. For task completions, there is NO via.

get_via_entity_type() → str
Get the entity type of the "via" of this notification.

The exact nature of this field is not yet known, but it seems to contain information about whatever caused this notification. For example, the chat message of an @mention, the topic reply for a reply, etc. For task completions, there is NO via.

get_via_id() → int
Get the ID of the "via" of this notification.

The exact nature of this field is not yet known, but it seems to contain information about whatever caused this notification. For example, the chat message of an @mention, the topic reply for a reply, etc. For task completions, there is NO via.

await set_status(*unread: bool*, *new: bool*) → None
Set the read/unread and seen/unseen (new) status of this notification.

This method sends requests.

Note: This also updates these properties in this object.

2.3.7 Creator

class pyryver.objects.**Creator**(*name: str*, *avatar: str*)
A message creator, with a name and an avatar.

This can be used to override the sender's display name and avatar.

Parameters

- **name** – The overriden display name
- **avatar** – The overriden avatar (a url to an image)

to_dict() → dict
Convert this Creator object to a dictionary to be used in a request.

Intended for internal use.

2.4 Utilities

2.4.1 Cache Data Storage

class pyryver.cache_storage.**AbstractCacheStorage**
Bases: abc.ABC

An abstract class defining the requirements for cache storages.

A cache storage is used by the Ryver class to cache chats data to improve performance.

abstractmethod load(*ryver: Ryver*, *obj_type: str*) → List[pyryver.objects.Object]
Load all saved objects of a specific type.

If no objects were saved, this method returns an empty list.

abstractmethod save(*obj_type: str, data: List[pyryver.objects.Object]*) → None
Save all objects of a specific type.

class pyryver.cache_storage.**FileCacheStorage**(*root_dir: str = '.'*, *prefix: str = ''*)
Bases: *pyryver.cache_storage.AbstractCacheStorage*

A cache storage implementation using files.

load(*ryver: Ryver*, *obj_type: str*) → List[pyryver.objects.Object]
Load all saved objects of a specific type.

If no objects were saved, this method returns an empty list.

save(*obj_type: str, data: List[pyryver.objects.Object]*) → None
Save all objects of a specific type.

2.4.2 API Helpers

async for ... in pyryver.util.**get_all**(*session: aiohttp.client.ClientSession*, *url: str*, *top: int = -1*, *skip: int = 0*, *param_sep: str = '?'*) → List[dict]
Because the REST API only gives 50 results at a time, this function is used to retrieve all objects.

Intended for internal use only.

pyryver.util.**get_type_from_entity**(*entity_type: str*) → str
Gets the object type from the entity type.

Note that it doesn't actually return a class, just the string.

Intended for internal use only.

await pyryver.util.**retry_until_available**(*coro: Awaitable[T], *args, timeout: float = None, **kwargs*) → T
Repeatedly tries to do some action (usually getting a resource) until the resource becomes available or a timeout elapses.

This function will try to run the given coroutine once every 0.5 seconds. If it results in a 404, the function tries again. Otherwise, the exception is raised.

If it times out, an `asyncio.TimeoutError` will be raised.

args and kwargs are passed to the coroutine.

Parameters

- **coro** – The coroutine to run
- **timeout** – The timeout in seconds, or None for no timeout

`pyryver.objects.get_obj_by_field`(*objs: List[pyryver.objects.Object], field: str, value: Any*) → pyryver.objects.Object

Gets an object from a list of objects by a field.

For example, this function can find a chat with a specific nickname in a list of chats.

Parameters

- **objs** – List of objects to search in.
- **field** – The field's name (usually a constant beginning with `FIELD_` in *pyryver.util*) within the object's JSON data.
- **value** – The value to look for.

2.4.3 Entity Types

`pyryver.util.TYPE_USER`
Corresponds to *`pyryver.objects.User`*.

`pyryver.util.TYPE_FORUM`
Corresponds to *`pyryver.objects.Forum`*.

`pyryver.util.TYPE_TEAM`
Corresponds to *`pyryver.objects.Team`*.

`pyryver.util.TYPE_GROUPCHAT_MEMBER`
Corresponds to *`pyryver.objects.GroupChatMember`*.

`pyryver.util.TYPE_TOPIC`
Corresponds to *`pyryver.objects.Topic`*.

`pyryver.util.TYPE_TOPIC_REPLY`
Corresponds to *`pyryver.objects.TopicReply`*.

`pyryver.util.TYPE_NOTIFICATION`
Corresponds to *`pyryver.objects.Notification`*.

`pyryver.util.TYPE_STORAGE`
Corresponds to *`pyryver.objects.Storage`*.

`pyryver.util.TYPE_FILE`
Corresponds to *`pyryver.objects.File`*.

2.4.4 Common Field Names

`pyryver.util.FIELD_USERNAME`

`pyryver.util.FIELD_EMAIL_ADDR`

`pyryver.util.`**`FIELD_DISPLAY_NAME`**
The object's display name (friendly name)

`pyryver.util.`**`FIELD_NAME`**

`pyryver.util.`**`FIELD_NICKNAME`**

`pyryver.util.`**`FIELD_ID`**
The object's ID, sometimes an `int`, sometimes a `str` depending on the object type.

`pyryver.util.`**`FIELD_JID`**
The object's JID, or JabberID. Used in the live socket interface for referring to chats.

Python Module Index

p

Index

I

L

M

N

O

P

R

S

T

U