pyryver

Release 0.4.0b1

Tyler Tian, Matthew Mirvish and Moeez Muhammad

Nov 03, 2020

CONTENTS

CHAPTER

ONE

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.

```
# Log in as a normal user
async with pyryver.Ryver("organization_name", "username", "password") as ryver:
    pass

# Log in with a token (for custom integrations)
async with pyryver.Ryver("organization_name", token="token") as ryver:
    pass
```

As the snippet above demonstrates, you can log in as a normal user, or using a token for a custom integration.

Warning: While both normal users and custom integrations can perform most actions, the Realtime API currently does not function when logging in with a token.

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_name", "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_name", "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))
```

(continues on next page)

(continued from previous page)

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:

Warning: The Realtime API currently does not work when logging in with a token.

```
async with pyryver.Ryver("organization_name", "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: pyryver.WSChatMessageData):
            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.terminate()`, which can be called from within event callbacks safely.

The contents of the `msg` parameter passed to our callback is an object of type `pyryver.ws_data.WSChatMessageData` that contains information about the message. In the `chat` message, there are two fields our "bot" needs to care about: `to_jid`, which specifies which chat the message was posted in, and `text`, which is the content of the message. `from_jid` refers to the message's creator. Perhaps unintuitively, the `to_jid` 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 JID here.

Note: Note that the callback will be called even if the message was sent by the current logged in user! Therefore, even if you want to respond to messages from everyone, you should still make sure to check that `from_jid` is not the bot user's JID to avoid replying to your own messages.

Notice how we're working with the chat's **JID** here, which is a string, as opposed to the regular ID, which is an integer. This is because 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_name", "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: pyryver.WSChatMessageData):
            # did the message come from a_user and was sent via DM to us?
            if msg.to_jid == me.get_jid() and msg.from_jid == 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_!")

        @session.on_connection_loss
        async def on_connection_loss():
            # Make sure that the session is terminated and run_forever() returns on␣
↪connection loss
            await session.terminate()

        await session.run_forever()
```

Note: Prior to v0.3.0, the `msg` parameter would have been a dict containing the raw JSON data of the message, and you would access the fields directly by name through dict lookups. If you still wish to access the raw data of the message, all message objects passed to callbacks have a `raw_data` attribute that contains the dict. In v0.3.2, `__getitem__()` was implemented for message objects to directly access the `raw_data` dict, providing (partial) backwards compatibility.

Here we also added a connection loss handler with the *`pyryver.ryver_ws.RyverWS.on_connection_loss()`* decorator. The connection loss handler calls `terminate()`, which causes `run_forever()` to return, allowing the program to exit on connection loss instead of waiting forever. Alternatively, you could also make the session auto-reconnect by doing `ryver.get_live_session(auto_reconnect=True)` when starting the session.

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

TWO

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: Optional[str] = None, user: Optional[str] = None, password: Optional[str] = None, token: Optional[str] = None, cache: Optional[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, it will be prompted using input(). If the username or password are not provided, and the token is not provided, the username and password will be prompted.

If a token is specified, the username and password will be ignored.

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 (optional). (as seen in the URL)
- **user** – The username to authenticate with (optional).
- **password** – The password to authenticate with (optional).
- **token** – The custom integration token to authenticate with (optional).
- **cache** – The aforementioned cache (optional).

async with get_live_session(*auto_reconnect: bool = False*) → *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.

Warning: Live sessions **do not work** when using a custom integration token.

Parameters **`auto_reconnect`** – Whether to automatically reconnect on connection loss.

Returns The live websockets session.

`get_chat`(***kwargs*) → Optional[*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.

Raises **`ValueError`** – If not all chats are loaded, or zero or multiple query parameters are specified.

Returns The chat, or None if not found.

`get_user`(***kwargs*) → Optional[*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
- name/display_name
- email

If using username or email to find the user, the search will be case-insensitive.

Returns None if not found.

Raises **`ValueError`** – If users are not loaded, or zero or multiple query parameters are specified.

Returns The user, or None of not found.

`get_forum`(***kwargs*) → Optional[*pyryver.objects.Forum*]

Get a specific forum.

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

Allowed query parameters are:

- id
- jid
- name
- nickname

If using nickname to find the chat, the search will be case-insensitive.

Returns None if not found.

Raises `ValueError` – If forums are not loaded, or zero or multiple query parameters are specified.

Returns The chat, or None if not found.

get_team(***kwargs*) → Optional[*pyryver.objects.Forum*]

Get a specific team.

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

Allowed query parameters are:

- id
- jid
- name
- nickname

If using nickname to find the chat, the search will be case-insensitive.

Returns None if not found.

Raises `ValueError` – If teams are not loaded, or zero or multiple query parameters are specified.

Returns The chat, or None if not found.

get_groupchat(*forums: bool = True, teams: bool = True, **kwargs*) → Optional[*pyryver.objects.GroupChat*]

Get a specific forum/team.

If no query parameters are supplied, more than one query parameters are supplied or the list to search is not loaded, raises `ValueError`.

Allowed query parameters are:

- id
- jid
- name
- nickname

If using nickname to find the chat, the search will be case-insensitive.

Returns None if not found.

Changed in version 0.4.0: Added parameters `forums` and `teams`.

Forums Whether to search the list of forums.

Teams Whether to search the list of teams.

Raises `ValueError` – If the list to search is not loaded, or zero or multiple query parameters are specified.

Returns The chat, or None if not found.

await get_object(*obj_type: Union[str, type], obj_id: Optional[int] = None, **kwargs*) → Union[Type[*pyryver.objects.Object*], List[Type[*pyryver.objects.Object*]]]

Get an object or multiple objects from Ryver with a type and optionally ID.

If extra keyword arguments are supplied, they are appended to the request as additional query parameters. Possible values include `top`, `skip`, `select`, `expand` and more. The Ryver Developer Docs or OData Specification contains documentation for some of these parameters. (Note: The link is to Odata 2.0 instead of 4.0 because the 2.0 page seems to be much more readable.)

With this method, you can get objects by properties other than ID. The following example gets one or more objects by display name:

```
# Note that this will return an array, even if there is only 1 result
user = await ryver.get_object(pyryver.User, filter=f"displayName eq '{name}'")
```

Parameters

- **obj_type** – The type of the object to retrieve, either a string type or the actual object type.
- **obj_id** – The object's ID (optional).

Raises `TypeError` – If the object is not instantiable.

Returns The object or list of objects requested.

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.

Returns The raw org and user info data.

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

Get the notifications for the logged in user.

Parameters

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

Returns An async iterator for the user's notifications.

await mark_all_notifs_read() → int

Marks all the user's notifications as read.

Returns How many notifications were marked as read.

await mark_all_notifs_seen() → int

Marks all the user's notifications as seen.

Returns How many notifications were marked as seen.

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

Upload a file to Ryver (for attaching to messages).

Note: Although this method uploads a file, the returned object is an instance of `Storage`, with type `Storage.TYPE_FILE`. 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()`.
- **`filetype`** – The MIME type of the file.

Returns The uploaded file, as a `Storage` object.

await create_link(*name: str*, *link_url: str*) → *pyryver.objects.Storage*

Create a link on Ryver (for attaching to messages).

Note: The returned object is an instance of `Storage` with type `Storage.TYPE_LINK`.

Parameters

- **`name`** – The name of this link (its title).
- **`url`** – The URL of this link.

Returns The created link, as a `Storage` object.

await invite_user(*email: str*, *role: str = 'member'*, *username: Optional[str] = None*, *display_name: Optional[str] = None*) → *pyryver.objects.User*

Invite a new user to the organization.

An optional username and display name can be specified to pre-populate those values in the User Profile page that the person is asked to fill out when they accept their invite.

Parameters

- **`email`** – The email of the user.
- **`role`** – The role of the user (member or guest), one of the `User.USER_TYPE_` constants (optional).
- **`username`** – The pre-populated username of this user (optional).
- **`display_name`** – The pre-populated display name of this user (optional).

Returns The invited user object.

await create_forum(*name: str, nickname: Optional[str] = None, about: Optional[str] = None, description: Optional[str] = None*) → *pyryver.objects.Forum*

Create a new open forum.

Parameters

- **name** – The name of this forum.
- **nickname** – The nickname of this forum (optional).
- **about** – The "about" (or "purpose" in the UI) of this forum (optional).
- **description** – The description of this forum (optional).

Returns The created forum object.

await create_team(*name: str, nickname: Optional[str] = None, about: Optional[str] = None, description: Optional[str] = None*) → *pyryver.objects.Team*

Create a new private team.

Parameters

- **name** – The name of this team.
- **nickname** – The nickname of this team (optional).
- **about** – The "about" (or "purpose" in the UI) of this team (optional).
- **description** – The description of this team (optional).

Returns The created team object.

await load_chats() → None

Load the data of all users/teams/forums.

This refreshes the cached data if a cache is supplied.

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.

await load_forums() → None

Load the data of all forums.

This refreshes the cached data if a cache is supplied.

await load_teams() → None

Load the data of all teams.

This refreshes the cached data if a cache is supplied.

await close()

Close this session.

2.2 Realtime Client

class `pyryver.ryver_ws.`**`RyverWS`**(*ryver: Ryver*, *auto_reconnect: bool = False*)

A live Ryver session using websockets.

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

> **Warning:** This **does not work** when using a custom integration token to sign in.

Parameters

- **`ryver`** – The `Ryver` object this live session came from.
- **`auto_reconnect`** – Whether to automatically reconnect on a connection loss.

`@`**`on_chat`**(*func: Callable[[pyryver.ws_data.WSChatMessageData], Awaitable]*)

Decorate a coroutine to be run when a new chat message is received.

This coroutine will be started as a task when a new chat message arrives. It should take a single argument of type `WSChatMessageData`, which contains the data for the message.

`@`**`on_chat_deleted`**(*func: Callable[[pyryver.ws_data.WSChatDeletedData], Awaitable]*)

Decorate a coroutine to be run when a chat message is deleted.

This coroutine will be started as a task when a chat message is deleted. It should take a single argument of type `WSChatDeletedData`, which contains the data for the message.

`@`**`on_chat_updated`**(*func: Callable[[pyryver.ws_data.WSChatUpdatedData], Awaitable]*)

Decorate a coroutine to be run when a chat message is updated (edited).

This coroutine will be started as a task when a chat message is updated. It should take a single argument of type `WSChatUpdatedData`, which contains the data for the message.

`@`**`on_presence_changed`**(*func: Callable[[pyryver.ws_data.WSPresenceChangedData], Awaitable]*)

Decorate a coroutine to be run when a user's presence changed.

This coroutine will be started as a task when a user's presence changes. It should take a single argument of type `WSPresenceChangedData`, which contains the data for the presence change.

`@`**`on_user_typing`**(*func: Callable[[pyryver.ws_data.WSUserTypingData], Awaitable]*)

Decorate a coroutine to be run when a user starts typing.

This coroutine will be started as a task when a user starts typing in a chat. It should take a single argument of type `WSUserTypingData`, which contains the data for the user typing.

`@`**`on_connection_loss`**(*func: Callable[[], Awaitable]*)

Decorate a coroutine to be run when the connection is lost.

This coroutine will be started as a task when the connection is lost. It should take no arguments.

A connection loss is determined using a ping task. A ping is sent to Ryver once every 10 seconds, and if the response takes over 5 seconds, this coroutine will be started. (These numbers roughly match those used by the official web client.)

If auto-reconnect is enabled, no action needs to be taken. Otherwise, applications are suggested to clean up and terminate, or try to reconnect using `RyverWS.try_reconnect()`. If *`RyverWS.run_forever()`* is used, *`RyverWS.terminate()`* should be called to make it return, unless you wish to reconnect.

A simple but typical implementation is shown below for applications that do not wish to recover:

```
async with ryver.get_live_session() as session:
    @session.on_connection_loss
    async def on_connection_loss():
        await session.terminate()
```

@**on_reconnect** (*func: Callable[], Awaitable]*)
Decorate a coroutine to be run when auto-reconnect succeeds.

This coroutine will be started as a task when auto-reconnect is successful. It should take no arguments. If auto-reconnect is not enabled, this coroutine will never be started.

EVENT_REACTION_ADDED = '/api/reaction/added'
A reaction was added to a message (includes topics, tasks and replies/comments).

`data` field format:

- `"type"`: The entity type of the thing that was reacted to.
- `"id"`: The ID of the thing that was reacted to. String for chat messages, int for everything else.
- `"userId"`: The ID of the user that reacted.
- `"reaction"`: The name of the emoji that the user reacted with.

EVENT_REACTION_REMOVED = '/api/reaction/removed'
A reaction was removed from a message (includes topics, tasks and replies/comments).

`data` field format:

- `"type"`: The entity type of the thing that was reacted to.
- `"id"`: The ID of the thing that was reacted to. String for chat messages, int for everything else.
- `"userId"`: The ID of the user that reacted.
- `"reaction"`: The name of the emoji that the user reacted with.

EVENT_TOPIC_CHANGED = '/api/activityfeed/posts/changed'
A topic was changed (created, updated, deleted).

`data` field format:

- `"created"`: A list of objects containing data for topics that were newly created.
- `"updated"`: A list of objects containing data for topics that were updated.
- `"deleted"`: A list of objects containing data for topics that were deleted.

EVENT_TASK_CHANGED = '/api/activityfeed/tasks/changed'
A task was changed (created, updated, deleted).

`data` field format:

- `"created"`: A list of objects containing data for tasks that were newly created.
- `"updated"`: A list of objects containing data for tasks that were updated.
- `"deleted"`: A list of objects containing data for tasks that were deleted.

EVENT_ENTITY_CHANGED = '/api/entity/changed'
Some entity was changed (created, updated, deleted).

`data` field format:

- `"change"`: The type of the change, could be "created", "updated", or "deleted".

- `"entity"`: The entity that was changed and some of its data after the change.

`EVENT_ALL = ''`
All unhandled events.

`@`**`on_event`**(*event_type: str*)
Decorate a coroutine to be run when an event occurs.

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 of type `WSEventData`, which contains the data for the event.

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

`MSG_TYPE_CHAT = 'chat'`
A chat message was received.

`MSG_TYPE_CHAT_UPDATED = 'chat_updated'`
A chat message was updated.

`MSG_TYPE_CHAT_DELETED = 'chat_deleted'`
A chat message was deleted.

`MSG_TYPE_PRESENCE_CHANGED = 'presence_change'`
A user changed their presence.

`MSG_TYPE_USER_TYPING = 'user_typing'`
A user is typing in a chat.

`MSG_TYPE_EVENT = 'event'`
An event occurred.

`MSG_TYPE_ALL = ''`
All unhandled messages.

`@`**`on_msg_type`**(*msg_type: str*)
Decorate a coroutine to be run when for a type of websocket messages or for all unhandled messages.

This coroutine will be started as a task when a new websocket 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 otherwise unhandled.

It should take a single argument of type `WSMessageData`, which contains the data for the event.

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 receive all otherwise unhandled messages.

`await send_chat`(*to_chat: Union[*pyryver.objects.Chat*, str], msg: str, timeout: float = 5.0*) → None
Send a chat message to a chat.

Parameters

- **`to_chat`** – The chat or the JID of the chat to send the message to.
- **`msg`** – The message contents.
- **`timeout`** – The timeout for waiting for an ack. If None, waits forever. By default waits for 5s.

Raises

- **`asyncio.TimeoutError`** – On ack timeout.
- ***`ClosedError`*** – If connection closed or not yet opened.

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

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

Useful for wrapping long running operations to make sure the typing indicator is kept, 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: Union[pyryver.objects.Chat, str], timeout: float = 5.0*) → None

Send a typing indicator to a chat.

The typing indicator automatically clears after a few seconds or when a message is sent. In private messages, you can also clear it with *RyverWS.send_clear_typing()* (does not work for group chats).

If you want to maintain the typing indicator for an extended operation, consider using *RyverWS.typing()*, which returns an async context manager that can be used to maintain the typing indicator for as long as desired.

Parameters

- **to_chat** – The chat or the JID of the chat to send the typing status to.
- **timeout** – The timeout for waiting for an ack. If None, waits forever. By default waits for 5s.

Raises

- **asyncio.TimeoutError** – On ack timeout.
- ***ClosedError*** – If connection closed or not yet opened.

await send_clear_typing(*to_chat: Union[pyryver.objects.Chat, str], timeout: float = 5.0*) → None

Clear the typing indicator for a chat.

Warning: For unknown reasons, this method **only works in private messages**.

Parameters

- **to_chat** – The chat or the JID of the chat to clear the typing status for.
- **timeout** – The timeout for waiting for an ack. If None, waits forever. By default waits for 5s.

Raises

- **asyncio.TimeoutError** – On ack timeout.
- ***ClosedError*** – If connection closed or not yet opened.

PRESENCE_AVAILABLE = 'available'

"Available" presence (green).

PRESENCE_AWAY = 'away'

"Away" presence (yellow clock).

PRESENCE_DO_NOT_DISTURB = 'dnd'
“Do Not Disturb” presence (red stop sign).

PRESENCE_OFFLINE = 'unavailable'
“Offline” presence (grey).

await send_presence_change(*presence: str, timeout: float = 5.0*) → None
Send a presence change message.

The presence change is global and not restricted to a single chat.

Parameters

- **presence** – The new presence, one of the `PRESENCE_` constants.
- **timeout** – The timeout for waiting for an ack. If None, waits forever. By default waits for 5s.

Raises

- **asyncio.TimeoutError** – On ack timeout.
- ***ClosedError*** – If connection closed or not yet opened.

is_connected() → bool
Get whether the websocket connection has been established.

Returns True if connected, False otherwise.

set_auto_reconnect(*auto_reconnect: bool*) → None
Set whether the live session should attempt to auto-reconnect on connection loss.

Parameters **auto_reconnect** – Whether to automatically reconnect.

await run_forever() → None
Run forever, or until `RyverWS.terminate()` is called.

Note: Since v0.4.0, this method will no longer return if `RyverWS.close()` is called. `RyverWS.terminate()` must be called instead, which closes the session if it is unclosed.

Note: By default, this method will only return if a fatal connection loss occurs and auto-reconnect is not enabled. If the connection loss is recoverable, this method will not return even if auto-reconnect is off.

You should use the `RyverWS.on_connection_loss()` decorator if you want to automatically close the connection and return on connection loss. See its documentation for an example.

await terminate() → None
Close the session and cause `RyverWS.run_forever()` to return.

This method will have no effect if called twice.

Note: If you use this class as an `async with` context manager, you don’t need to call the following two methods.

await start(*timeout: float = 5.0*) → None
Start the session, or reconnect after a connection loss.

Note: If there is an existing connection, it will be closed.

Parameters **`timeout`** – The connection timeout in seconds. If None, waits forever. By default, waits for 5 seconds.

`await close`(*cancel_rx: bool = True*, *cancel_ping: bool = True*) → None

Close the session.

Any future operation after closing will result in a `ClosedError` being raised, unless the session is reconnected using `RyverWS.start()` or `RyverWS.try_reconnect()`.

When used as an async context manager, this method does not need to be called.

Note: Since v0.4.0, this method no longer causes `RyverWS.run_forever()` to return. Use `RyverWS.terminate()` if you want to close the session and exit `run_forever()`.

Parameters

- **`cancel_rx`** – Whether to cancel the rx task. For internal use only.
- **`cancel_ping`** – Whether to cancel the ping task. For internal use only.

`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`() → None

Start sending the typing indicator.

`await stop`() → None

Stop sending the typing indicator.

Note: This method will attempt to clear the typing indicator using `RyverWS.send_clear_typing()`. However, it only works in private messages. Outside of private messages, 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.`**`WSConnectionError`**

Bases: `Exception`

An exception raised by the real-time websockets session to indicate some kind of connection issue.

`class` `pyryver.ryver_ws.`**`ClosedError`**

Bases: `pyryver.ryver_ws.WSConnectionError`

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

`class` `pyryver.ryver_ws.`**`ConnectionLossError`**

Bases: `pyryver.ryver_ws.WSConnectionError`

An exception raised to indicate that the connection was lost in the middle of an operation.

2.2.1 Callback Task Data Types

class pyryver.ws_data.**WSMessageData**(*ryver: Ryver*, *data: dict*)

The data for any websocket message in *pyryver.ryver_ws.RyverWS*.

Variables

- **ryver** – The Ryver session that the data came from.
- **ws_msg_type** – The type of this websocket message. This can be one of the `MSG_TYPE_` constants in *pyryver.ryver_ws.RyverWS* (except `MSG_TYPE_ALL`). However, do note that the constants listed there do not cover all valid values of this field.
- **raw_data** – The raw websocket message data.

class pyryver.ws_data.**WSChatMessageData**(*ryver: Ryver*, *data: dict*)

Bases: *pyryver.ws_data.WSMessageData*

The data for a chat message in *pyryver.ryver_ws.RyverWS*.

Variables

- **message_id** – The ID of the message (a string).
- **from_jid** – The JID of the sender of this message.
- **to_jid** – The JID of the chat this message was sent to.
- **text** – The contents of the message.
- **subtype** – The subtype of the message. This will be one of the `SUBTYPE_` constants in `ChatMessage`.
- **attachment** – The file attached to this message, or None if there isn't one.
- **creator** – The overridden message creator (see `Creator`), or None if there isn't one.

class pyryver.ws_data.**WSChatUpdatedData**(*ryver: Ryver*, *data: dict*)

Bases: *pyryver.ws_data.WSChatMessageData*

The data for a chat message edited in *pyryver.ryver_ws.RyverWS*.

Variables

- **message_id** – The ID of the message (a string).
- **from_jid** – The JID of the user that edited the message.
- **to_jid** – The JID of the chat this message was sent to.
- **text** – The contents of the message after the edit. Note: In very rare circumstances, this field is known to be `None`.
- **subtype** – The subtype of the message. This will be one of the `SUBTYPE_` constants in `ChatMessage`.
- **attachment** – The file attached to this message, or None if there isn't one.

class pyryver.ws_data.**WSChatDeletedData**(*ryver: Ryver*, *data: dict*)

Bases: *pyryver.ws_data.WSChatMessageData*

The data for a chat message deleted in *pyryver.ryver_ws.RyverWS*.

Variables

- **message_id** – The ID of the message (a string).
- **from_jid** – The JID of the sender of this message.

- **`to_jid`** – The JID of the chat this message was sent to.
- **`text`** – The contents of the message that was deleted.
- **`subtype`** – The subtype of the message. This will be one of the `SUBTYPE_` constants in `ChatMessage`.
- **`attachment`** – The file attached to this message, or None if there isn't one.

class `pyryver.ws_data.`**`WSPresenceChangedData`**(*ryver: Ryver*, *data: dict*)

Bases: *`pyryver.ws_data.WSMessageData`*

The data for a presence changed in *`pyryver.ryver_ws.RyverWS`*.

Variables

- **`presence`** – The new presence. This will be one of the `PRESENCE_` constants in `RyverWS`.
- **`from_jid`** – The JID of the user that changed their presence.
- **`client`** – The client the user is using.
- **`timestamp`** – An ISO 8601 timestamp of this event. You can use *`pyryver.util.iso8601_to_datetime()`* to convert it into a datetime.

class `pyryver.ws_data.`**`WSUserTypingData`**(*ryver: Ryver*, *data: dict*)

Bases: *`pyryver.ws_data.WSMessageData`*

The data for a user typing in *`pyryver.ryver_ws.RyverWS`*.

Variables

- **`from_jid`** – The JID of the user that started typing.
- **`to_jid`** – The JID of the chat the user started typing in.
- **`state`** – The "state" of the typing. This is almost always "composing" (for typing in progress), but it could also very rarely be "done", for when the user has finished typing.

class `pyryver.ws_data.`**`WSEventData`**(*ryver: Ryver*, *data: dict*)

Bases: *`pyryver.ws_data.WSMessageData`*

The data for an event in *`pyryver.ryver_ws.RyverWS`*.

Variables

- **`event_type`** – The type of this event. This can be one of the `EVENT_` constants in *`pyryver.ryver_ws.RyverWS`* (except `EVENT_ALL`). However, do note that the constants listed there do not cover all valid values of this field.
- **`event_data`** – The data of this event. This is a dictionary mapping strings to values of any type. The format depends on the event type. The format of some events are documented in the docs of the `EVENT_` constants.

2.3 Data Models

class `pyryver.objects.`**`Object`**`(`*ryver: pyryver.ryver.Ryver*, *data: dict*`)`

Base class for all Ryver objects.

Parameters

- **`ryver`** – The parent `pyryver.pyryver.Ryver` instance.
- **`data`** – The object's data.

`get_ryver``()` → *pyryver.ryver.Ryver*

Get the Ryver session this object was retrieved from.

Returns The Ryver session associated with this object.

`get_id``()` → Union[int, str]

Get the ID of this object.

For a *`ChatMessage`* this is a string. For all other types, it is an int.

Returns The ID of this object.

`get_entity_type``()` → str

Get the entity type of this object.

Returns The entity type of this object, or if no entity of such type exists, `<unknown>`.

`get_raw_data``()` → dict

Get the raw data of this object.

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

Returns The raw data of this object.

`get_api_url``(`**args*, ***kwargs*`)` → str

Uses `Ryver.get_api_url()` to get a URL for working with the Ryver API.

Warning: This method is intended for internal use only.

This is equivalent to calling `Ryver.get_api_url()`, but with the first two parameters set to `self.get_type()` and `self.get_id()`.

Returns The formatted URL for performing requests.

`get_create_date``()` → Optional[str]

Get the date this object was created as an ISO 8601 timestamp.

Note: This method does not work for all objects. For some objects, it will return None.

Tip: You can use *`pyryver.util.iso8601_to_datetime()`* to convert the timestamps returned by this method into a datetime.

Returns The creation date of this object, or None if not supported.

`get_modify_date``()` → Optional[str]

Get the date this object was last modified as an ISO 8601 timestamp.

Note: This method does not work for all objects. For some objects, it will return None.

Tip: You can use *`pyryver.util.iso8601_to_datetime()`* to convert the timestamps returned by this method into a datetime.

Returns The modification date of this object, or None if not supported.

`get_app_link``()` → str

Get a link to this object that opens the app to this object.

Note: This method does not work for some types such as messages and topic/task replies. Additionally, only types with *`Object.is_instantiable()`* true can be linked to. Calling this method on an object of an invalid type will result in a `TypeError`.

Raises **`TypeError`** – If this object cannot be linked to.

Returns The in-app link for this object.

`get_creator``()` → Optional[*pyryver.objects.Creator*]

Get the Creator of this object.

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.

Not all objects support this operation. If not supported, this will return `None`.

Returns The overridden creator of this message.

`await get_deferred_field`(*field: str*, *field_type: str*) → Union[Type[*pyryver.objects.Object*], List[Type[*pyryver.objects.Object*]]]

Get the value of a field of this object that exists, but is not included ("__deferred" in the Ryver API).

Warning: This function is intended for internal use only.

This function will automatically infer from the result's contents whether to return a single object or a list of objects.

If the field cannot be retrieved, a `ValueError` will be raised.

Parameters

- **`field`** – The name of the field.
- **`field_type`** – The type of the field, must be a `TYPE_` constant.

Returns The expanded value of this field as an object or a list of objects.

Raises **`ValueError`** – When the field cannot be expanded.

await get_create_user() → Optional[*pyryver.objects.User*]

Get the user that created this object.

> **Note:** This method does not work for all objects. If not supported, it will return None.

Returns The user that created this object, or None if not supported.

await get_modify_user() → Optional[*pyryver.objects.User*]

Get the user that modified this object.

> **Note:** This method does not work for all objects. If not supported, it will return None.

Returns The user that last modified this object, or None if not supported.

classmethod get_type() → str

Get the type of this object.

Returns The type of this object.

classmethod is_instantiable() → bool

Get whether this object type is instantiable.

Some types of objects cannot be instantiated, as they are actually not a part of the REST API, such as `Message`, `Chat`, and other abstract types. If the type can be instantiated, this class method will return `True`.

Note that even though a type may not be instantiable, its derived types could still be. For example, `Chat` is not instantiable, but one of its derived types, `User`, is instantiable.

Returns Whether this type is instantiable.

classmethod await get_by_id(*ryver:* pyryver.ryver.Ryver, *obj_id:* *int*) → Type[*pyryver.objects.Object*]

Retrieve an object of this type by ID.

Parameters

- **ryver** – The Ryver session to retrieve the object from.
- **obj_id** – The ID of the object to retrieve.

Raises **TypeError** – If this type is not instantiable.

Returns The object requested.

2.3.1 Chats

class pyryver.objects.**Chat**(*ryver:* pyryver.ryver.Ryver, *data:* *dict*)

Bases: `pyryver.objects.Object`

Any Ryver chat you can send messages to.

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

get_jid() → str

Get the JID (JabberID) of this chat.

The JID is used in the websockets interface.

Returns The JID of this chat.

abstractmethod get_name () → str

Get the name of this chat.

Returns The name of this chat.

get_task_tags () → List[pyryver.objects.TaskTag]

Get a list of task tags defined in this chat, as `TaskTag` objects.

Returns The defined task tags of this chat.

await set_task_tags (*tags: Iterable[pyryver.objects.TaskTag]*)

Set the task tags defined in this chat.

Note: This will erase any existing tags.

This method also updates the task tags property of this object.

Parameters tags – The new tags as a list of ``TaskTag``s.

await send_message (*message: str, creator: Optional[pyryver.objects.Creator] = None, attachment: Union[Storage, File, None] = None, from_user: Optional[User] = None*) → str

Send a message to this chat.

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

Tip: To attach a file to the message, use *`pyryver.ryver.Ryver.upload_file()`* to upload the file you wish to attach. Alternatively, use *`pyryver.ryver.Ryver.create_link()`* for a link attachment.

Warning: `from_user` **must** be set when using attachments with private messages. Otherwise, a `ValueError` will be raised. It should be set to the user that is sending the message (the user currently logged in).

It is not required to be set if the message is being sent to a forum/team.

Returns the ID of the chat message sent (**not** the message object itself). Note that message IDs are strings.

Parameters

- **message** – The message contents.
- **creator** – The overridden creator; optional, if unset uses the logged-in user's profile.
- **attachment** – An attachment for this message, e.g. a file or a link (optional). Can be either a `Storage` or a `File` object.
- **from_user** – The user that is sending this message (the user currently logged in); **must** be set when using attachments in private messages (optional).

Raises ValueError – If a `from_user` is not provided for a private message attachment.

Returns The ID of the chat message that was sent.

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.

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.

Returns An async iterator for the topics of this chat.

await get_messages(*count: int*, *skip: int = 0*) → List[*pyryver.objects.ChatMessage*]

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

Parameters

- **count** – Maximum number of results.
- **skip** – The number of results to skip (optional).

Returns A list of messages.

await get_message(*msg_id: str*) → *pyryver.objects.ChatMessage*

Get a single message from this chat by its ID.

Note: There is a chance that this method might result in a 404 Not Found for messages that were sent recently (such as when using the realtime websocket API (`pyryver.ryver_ws.RyverWS`) to respond to messages), as those messages have not been fully added to Ryver's database yet.

You can use `pyryver.util.retry_until_available()` to wrap around this coroutine to get around this.

Parameters **msg_id** – The ID of the chat message to get.

Returns The message object.

await get_messages_surrounding(*msg_id: str*, *before: int = 0*, *after: int = 0*) → List[*pyryver.objects.ChatMessage*]

Get a range of messages (most recent **last**) before and after a chat message (given by ID).

Warning: Before and after cannot exceed 25 messages, otherwise a `aiohttp.ClientResponseError` will be raised with the code 400 Bad Request.

Note: There is a chance that this method might result in a 404 Not Found for messages that were sent recently (such as when using the realtime websocket API (`pyryver.ryver_ws.RyverWS`) to respond to messages), as those messages have not been fully added to Ryver's database yet.

You can use `pyryver.util.retry_until_available()` to wrap around this coroutine to get around this.

The message with the given ID is also included as a part of the result.

Parameters

- **msg_id** – The ID of the message to use as the reference point.
- **before** – How many messages to retrieve before the specified one (optional).
- **after** – How many messages to retrieve after the specified one (optional).

Returns The messages requested, including the reference point message.

await get_task_board() → Optional[*pyryver.objects.TaskBoard*]
Get the task board of this chat.

If tasks are not set up for this chat, this will return None.

This method works on users too. If used on a user, it will get their personal task board.

Returns The task board of this chat.

await delete_task_board() → bool
Delete (or "reset", according to the UI) the task board of this chat.

This method will not yield an error even if there is no task board set up. In those cases, it will simply return false.

Returns Whether the task board was deleted.

await create_task_board(*board_type: str, prefix: Optional[str] = None, categories: Optional[List[Union[str, Tuple[str, str]]]] = None, uncategorized_name: Optional[str] = None*) → *pyryver.objects.TaskBoard*
Create the task board for this chat if it has not yet been set up.

The board type should be one of the `TaskBoard` `BOARD_TYPE_` constants; it specified whether this task board should be a simple list or a board with categories.

You can also specify a list of category names and optional category types to pre-populate the task board with categories. Each entry in the list should either be a string, which specifies the category name, or a tuple of the name and the type of the category (a `CATEGORY_TYPE_` constant). The default category type is `TaskCategory.CATEGORY_TYPE_OTHER`.

An "uncategorized" category is always automatically added. Therefore, the type `TaskCategory.CATEGORY_TYPE_UNCATEGORIZED` cannot be used in the list. You can, however, change the name of the default "Uncategorized" category by specifying `uncategorized_name`.

Categories should not be specified if the type of the task board is `TaskBoard.BOARD_TYPE_LIST`.

Parameters

- **board_type** – The type of the task board.
- **prefix** – The task prefix (optional).
- **categories** – A list of categories and optional types to pre-populate the task board with (see above) (optional).
- **uncategorized_name** – The name for the default "Uncategorized" category.

await delete_avatar() → None
Delete the avatar of this chat.

await set_avatar(*filename: str, filedata: Any, filetype: Optional[str] = None*) → None
Set the avatar of this chat.

A wrapper for `Storage.make_avatar_of()` and `Ryver.upload_file()`.

Parameters

- **filename** – The filename of the image.

- **filedata** – The image's raw data, sent directly to `aiohttp.FormData.add_field()`.
- **filetype** – The MIME type of the file.

class pyryver.objects.**GroupChat**(*ryver: pyryver.ryver.Ryver*, *data: dict*)
Bases: *pyryver.objects.Chat*

A Ryver team or forum.

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

Returns The name of this forum/team.

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

The nickname is a unique identifier that can be used to refer to the chat across Ryver.

Returns The nickname of this forum/team.

has_chat() → bool
Get whether this forum/team has a chat tab.

Returns Whether there is a chat tab for this forum/team.

has_topics() → bool
Get whether this forum/team has a topics tab.

Returns Whether there is a topics tab for this forum/team.

has_tasks() → bool
Get whether this forum/team has a tasks tab.

Returns Whether there is a tasks tab for this forum/team.

does_announce_topics() → bool
Get whether new topics are announced with a chat message.

Returns Whether new topics are announced with a chat message.

does_announce_tasks() → bool
Get whether new tasks are announced with a chat message.

Returns Whether new tasks are announced with a chat message.

is_archived() → bool
Get whether this team/forum is archived.

Returns Whether the team/forum is archived.

async for ... in get_members(*top: int = -1*, *skip: int = 0*) → AsyncIterator[*pyryver.objects.GroupChatMember*]
Get all the members of this chat.

Note: This gets the members as *GroupChatMember* objects, which contain additional info such as whether the user is an admin of this chat.

To get the *User* object, use *GroupChatMember.as_user()*.

Parameters

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

- **`skip`** – Skip this many results.

Returns An async iterator for the members of this chat.

await get_member(*user: Union[int, pyryver.objects.User]*) → Optional[*pyryver.objects.GroupChatMember*]

Get a member using either a User object or user ID.

Note: This gets the member as a *`GroupChatMember`* object, which contain additional info such as whether the user is an admin of this chat.

Note: If an ID is provided, it should be the **user** ID of this member, not the groupchat member ID.

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

Parameters **`user`** – The user, or the ID of the user.

Returns The member, or None if not found.

await add_member(*user: pyryver.objects.User, role: str = None*) → None

Add a member to this forum or team.

This is a wrapper for *`User.add_to_chat()`*.

The `role` should be either *`GroupChatMember.ROLE_MEMBER`* or *`GroupChatMember.ROLE_ADMIN`*. By default, new members are invited as normal members.

Parameters

- **`user`** – The user to add.
- **`role`** – The role to invite the user as (member or admin) (optional).

await remove_member(*user: Union[int, pyryver.objects.User]*) → None

Remove a user from this forum/team.

Either a user ID or a user object can be used.

If the user is not in this forum/team, this method will do nothing.

Parameters **`user`** – The user or the ID of the user to remove.

await create_topic(*subject: str, body: str, stickied: bool = False, attachments: Optional[Iterable[Union[Storage, File]]] = None, creator: Optional[pyryver.objects.Creator] = None*) → *pyryver.objects.Topic*

Create a topic in this chat.

Returns the topic created.

Tip: To attach files to the topic, use *`pyryver.ryver.Ryver.upload_file()`* to upload the files you wish to attach. Alternatively, use *`pyryver.ryver.Ryver.create_link()`* for link attachments.

Changed in version 0.4.0: Switched the order of attachments and creator for consistency.

Parameters

- **`subject`** – The subject (or title) of the new topic.
- **`body`** – The contents of the new topic.

- **stickied** – Whether to sticky (pin) this topic to the top of the list (optional, default False).
- **attachments** – A number of attachments for this topic (optional).
- **creator** – The overridden creator; optional, if unset uses the logged-in user's profile.

Returns The created topic.

await change_settings(*chat: Optional[bool] = NO_CHANGE, topics: Optional[bool] = NO_CHANGE, tasks: Optional[bool] = NO_CHANGE, announce_topics: Optional[bool] = NO_CHANGE, announce_tasks: Optional[bool] = NO_CHANGE*) → None

Change the settings of this forum/team.

Note: The settings here contain only the settings in the "Settings" tab in the UI.

This method also updates these properties in this object.

If any parameters are unspecified, `NO_CHANGE`, or `None`, they will be left as-is.

Parameters

- **chat** – Whether there should be a chat tab for this forum/team (optional).
- **topics** – Whether there should be a topics tab for this forum/team (optional).
- **tasks** – Whether there should be a tasks tab for this form/team (optional).
- **announce_topics** – Whether new topics should be announced in the chat (optional).
- **announce_tasks** – Whether new tasks should be announced in the chat (optional).

await set_archived(*archived: bool*) → None

Set whether this team/forum is archived.

Note: This method also updates the archived property of this object.

Parameters **archived** – Whether this team/forum should be archived.

await delete() → None

Delete this forum/team.

As with most things, once it's deleted, there's no way to go back!

await join() → None

Join this forum/team as the current logged in user.

await leave() → None

Leave this forum/team as the current logged in user.

Note: This is not the same as selecting "Close and keep closed" in the UI. With this, the user will no longer show up in the members list of the forum/team, whereas "Close and keep closed" will still keep the user in the forum/team and only update the notification settings.

class pyryver.objects.**Forum**(*ryver:* pyryver.ryver.Ryver, *data: dict*)

Bases: *pyryver.objects.GroupChat*

A Ryver forum.

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

A Ryver team.

2.3.2 Users

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

A Ryver user.

Variables

- ***ROLE_USER*** – Regular organization member. Admins also have this role in addition to `ROLE_ADMIN`.
- ***ROLE_ADMIN*** – An org admin.
- ***ROLE_GUEST*** – A guest.
- ***USER_TYPE_MEMBER*** – A member.
- ***USER_TYPE_GUEST*** – A guest.

ROLE_USER = 'ROLE_USER'

ROLE_ADMIN = 'ROLE_ADMIN'

ROLE_GUEST = 'ROLE_GUEST'

USER_TYPE_MEMBER = 'member'

USER_TYPE_GUEST = 'guest'

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

Returns The username of this user.

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

Returns The display name of this user.

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

Returns The 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).

Returns The user's "Role" as described in their profile.

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

Returns The user's "About" as described in their profile.

get_time_zone() → str

Get this user's Time Zone.

Returns The user's time zone.

get_email_address() → str

Get this user's Email Address.

Returns The user's email address.

get_activated() → bool

Get whether this user's account is activated.

Returns Whether this user's account is activated (enabled).

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.

Returns The user's roles in the organization.

get_user_type() → str

Get the type of this user (member or guest).

The returned value will be either `User.USER_TYPE_MEMBER` or `User.USER_TYPE_GUEST`.

Returns The type of the user.

is_admin() → bool

Get whether this user is an org admin.

Returns Whether the user is an org admin.

accepted_invite() → bool

Get whether this user has accepted their user invite.

Returns Whether the user has accepted their invite.

await set_profile(*display_name: Optional[str] = None*, *role: Optional[str] = None*, *about: Optional[str] = None*) → None

Update this user's profile.

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

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.

await set_activated(*activated: bool*) → None

Activate or deactivate the user. Requires admin.

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`.

Note: Although for org admins, `get_roles()` will return both `ROLE_USER` and `ROLE_ADMIN`, to make someone an org admin you only need to pass `ROLE_ADMIN` into this method.

Note: This also updates these properties in this object.

await add_to_chat(*chat:* pyryver.objects.GroupChat, *role: Optional[str] = None*) → None

Add a user to a forum/team.

The `role` should be either `GroupChatMember.ROLE_MEMBER` or `GroupChatMember.ROLE_ADMIN`. By default, new members are invited as normal members.

Parameters

- **chat** – The forum/team to add to.
- **role** – The role to invite the user as (member or admin) (optional, defaults to member).

await create_topic(*from_user:* pyryver.objects.User, *subject: str*, *body: str*, *stickied: bool = False*, *attachments: Optional[Iterable[Union[Storage, File]]] = None*, *creator: Optional[*pyryver.objects.Creator*] = None*) → *pyryver.objects.Topic*

Create a topic in this user's DMs.

Returns the topic created.

Tip: To attach files to the topic, use `pyryver.ryver.Ryver.upload_file()` to upload the files you wish to attach. Alternatively, use `pyryver.ryver.Ryver.create_link()` for link attachments.

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 new topic.
- **body** – The contents of the new topic.
- **stickied** – Whether to sticky (pin) this topic to the top of the list (optional, default False).
- **attachments** – A number of attachments for this topic (optional).
- **creator** – The overridden creator; optional, if unset uses the logged-in user's profile.

Returns The created topic.

2.3.3 Group Chat Members

class `pyryver.objects.`**`GroupChatMember`**(*ryver: pyryver.ryver.Ryver*, *data: dict*)

Bases: *`pyryver.objects.Object`*

A member in a forum or team.

This class can be used to tell whether a user is an admin of their forum/team.

Variables

- ***`ROLE_MEMBER`*** – Regular chat member. Note: This member could also be an org admin.
- ***`ROLE_ADMIN`*** – Forum/team admin.

`ROLE_MEMBER = 'ROLE_TEAM_MEMBER'`

`ROLE_ADMIN = 'ROLE_TEAM_ADMIN'`

`get_role``()` → str

Get the role of this member.

This will be one of the `ROLE_` constants in this class.

Returns The role of this member.

`await as_user``()` → *pyryver.objects.User*

Get this member as a *`User`* object.

Returns The member as a `User` object.

`get_name``()` → str

Get the display name of this member.

`is_admin``()` → bool

Get whether this member is an admin of their forum/team.

Warning: This method does not check for org admins.

Returns Whether this user is a forum admin/team admin.

`await remove``()` → None

Remove this member from the forum/team.

`await set_role``(`*role: str*`)` → None

Set the role of this member (regular member or admin).

The role should be one of the `ROLE_` constants in this class.

Note: This will also update the role stored in this object.

Parameters **`role`** – The new role of the member.

2.3.4 Messages (Including Topics)

class `pyryver.objects.`**`Message`**(*ryver:* pyryver.ryver.Ryver, *data:* *dict*)

Bases: *`pyryver.objects.Object`*

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

`get_body`() → str

Get the body of this message.

Note that this may be None in some circumstances.

Returns The body of this message.

await get_author() → *pyryver.objects.User*

Get the author of this message, as a *`User`* object.

Returns The author of this message.

await react(*emoji:* *str*) → None

React to this message with an emoji.

Note: This method does **not** update the reactions property of this object.

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

await unreact(*emoji:* *str*) → None

Unreact with an emoji.

Note: This method does **not** update the reactions property of this object.

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

`get_reactions`() → dict

Get the reactions on this message.

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

Returns A dict matching each emoji to the users that reacted with that emoji.

`get_reaction_counts`() → dict

Count the number of reactions for each emoji on a message.

Returns a dict of `{emoji:  number_of_reacts}`.

Returns A dict matching each emoji to the number of users that reacted with that emoji.

await delete() → None

Delete this message.

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

Bases: *`pyryver.objects.Message`*

A message that was sent to a chat.

Note: Chat message are actually not a part of the Ryver REST APIs, since they aren't standalone objects (a chat is required to obtain one). As a result, they are a bit different from the other objects. Their IDs are strings rather than ints, and they are not instantiable (and therefore cannot be obtained from `Ryver.get_object()` or *Object.get_by_id()*.)

Variables

- ***MSG_TYPE_PRIVATE*** – A private message between users.
- ***MSG_TYPE_GROUPCHAT*** – A message sent to a group chat (team or forum).
- ***SUBTYPE_CHAT_MESSAGE*** – A regular chat message sent by a user.
- ***SUBTYPE_TOPIC_ANNOUNCEMENT*** – An automatic chat message that announces the creation of a new topic.
- ***SUBTYPE_TASK_ANNOUNCEMENT*** – An automatic chat message that announces the creation of a new task.

MSG_TYPE_PRIVATE = 'chat'

MSG_TYPE_GROUPCHAT = 'groupchat'

SUBTYPE_CHAT_MESSAGE = 'chat'

SUBTYPE_TOPIC_ANNOUNCEMENT = 'topic_share'

SUBTYPE_TASK_ANNOUNCEMENT = 'task_share'

get_msg_type() → str

Get the type of this message (private message or group chat message).

The returned value will be one of the `MSG_TYPE_` constants in this class.

Returns The type of this message.

get_subtype() → str

Get the subtype of this message (regular message or topic/task announcement).

The returned value will be one of the `SUBTYPE_` constants in this class.

Returns The subtype of this message.

get_time() → str

Get the time this message was sent, as an ISO 8601 timestamp.

Tip: You can use *pyryver.util.iso8601_to_datetime()* to convert the timestamps returned by this method into a datetime.

Returns The time this message was sent, as an ISO 8601 timestamp.

get_author_id() → int

Get the ID of the author of this message.

Returns The author ID of the message.

get_chat_type() → str

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

Returns The type of the chat this message was sent to.

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.

Returns The ID of the chat this message was sent to.

get_attached_file() → Optional[*pyryver.objects.File*]

Get the file attached to this message, if there is one.

Note: 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.

Note: Even if the attachment was a link and not a file, it will still be returned as a `File` object, as there seems to be no way of telling the type of the attachment just from the info provided in the message object.

Returns None otherwise.

Returns The attached file or link.

get_announced_topic_id() → Optional[int]

Get the ID of the topic this message is announcing.

This is only a valid operation for messages that announce a new topic. In other words, *ChatMessage.get_subtype()* must return *ChatMessage.SUBTYPE_TOPIC_ANNOUNCEMENT*. If this message does not announce a topic, this method will return `None`.

Returns The ID of the topic that is announced by this message, or `None`.

get_announced_task_id() → Optional[int]

Get the ID of the task this message is announcing.

This is only a valid operation for messages that announce a new task. In other words, *ChatMessage.get_subtype()* must return *ChatMessage.SUBTYPE_TASK_ANNOUNCEMENT*. If this message does not announce a topic, this method will return `None`.

Returns The ID of the task that is announced by this message, or `None`.

await get_author() → *pyryver.objects.User*

Get the author of this message, as a *User* object.

Tip: For chat messages, you can get the author ID without sending any requests, with *ChatMessage.get_author_id()*.

Returns The author of this message.

await get_chat() → *pyryver.objects.Chat*

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

Returns The chat this message was sent to.

await react(*emoji: str*) → None

React to this task with an emoji.

Note: This method does **not** update the reactions property of this object.

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

await unreact(*emoji: str*) → None

Unreact with an emoji.

Note: This method does **not** update the reactions property of this object.

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

await delete() → None

Deletes the message.

await edit(*message: Optional[str] = NO_CHANGE, creator: Optional[pyryver.objects.Creator] = NO_CHANGE, attachment: Union[Storage, File, None] = NO_CHANGE, from_user: Optional[User] = None*) → None

Edit this message.

Note: You can only edit a message if it was sent by you (even if you are an admin). Attempting to edit another user's message will result in a `aiohttp.ClientResponseError`.

This also updates these properties in this object.

Tip: To attach a file to the message, use *`pyryver.ryver.Ryver.upload_file()`* to upload the file you wish to attach. Alternatively, use *`pyryver.ryver.Ryver.create_link()`* for a link attachment.

Warning: `from_user` **must** be set when using attachments with private messages. Otherwise, a `ValueError` will be raised. It should be set to the user that is sending the message (the user currently logged in).

It is not required to be set if the message is being sent to a forum/team.

If any parameters are unspecified or `NO_CHANGE`, they will be left as-is. Passing `None` for parameters for which `None` is not a valid value will also result in the value being unchanged.

Parameters

- **`message`** – The new message contents (optional).
- **`creator`** – The new creator (optional).
- **`attachment`** – An attachment for this message, e.g. a file or a link (optional). Can be either a `Storage` or a `File` object.
- **`from_user`** – The user that is sending this message (the user currently logged in); **must** be set when using attachments in private messages (optional).

Raises **`ValueError`** – If a `from_user` is not provided for a private message attachment.

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

Bases: `pyryver.objects.PostedMessage`

A Ryver topic in a chat.

`get_subject`() → str

Get the subject of this topic.

Returns The subject of this topic.

`is_stickied`() → bool

Return whether this topic is stickied (pinned) to the top of the list.

Returns Whether this topic is stickied.

`is_archived`() → bool

Return whether this topic is archived.

Returns Whether this topic is archived.

await `archive`(*archived: bool = True*) → None

Archive or un-archive this topic.

Parameters **`archived`** – Whether the topic should be archived.

await `unarchive`() → None

Un-archive this topic.

This is the same as calling *`Topic.archive()`* with False.

await `reply`(*message: str*, *creator: Optional[pyryver.objects.Creator] = None*, *attachments: Optional[Iterable[Union[Storage, File]]] = None*) → *pyryver.objects.TopicReply*

Reply to the topic.

Note: For unknown reasons, overriding the creator does not seem to work for this method.

Tip: To attach files to the reply, use *`pyryver.ryver.Ryver.upload_file()`* to upload the files you wish to attach. Alternatively, use *`pyryver.ryver.Ryver.create_link()`* for link attachments.

Parameters

- **`message`** – The reply content
- **`creator`** – The overridden creator (optional). **Does not work.**
- **`attachments`** – A number of attachments for this reply (optional).

Returns The created reply.

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

Get all the replies to this topic.

Parameters

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

Returns An async iterator for the replies of this topic.

await edit(*subject: Optional[str] = NO_CHANGE, body: Optional[str] = NO_CHANGE, stickied: Optional[bool] = NO_CHANGE, creator: Optional[pyryver.objects.Creator] = NO_CHANGE, attachments: Optional[Iterable[Union[Storage, File]]] = NO_CHANGE*) → None

Edit this topic.

Note: Unlike editing topic replies and chat messages, admins have permission to edit any topic regardless of whether they created it.

The file attachments (if specified) will **replace** all existing attachments.

Additionally, this method also updates these properties in this object.

If any parameters are unspecified or `NO_CHANGE`, they will be left as-is. Passing `None` for parameters for which `None` is not a valid value will also result in the value being unchanged.

Parameters

- **subject** – The subject (or title) of the topic (optional).
- **body** – The contents of the topic (optional).
- **stickied** – Whether to sticky (pin) this topic to the top of the list (optional).
- **creator** – The overridden creator (optional).
- **attachments** – A number of attachments for this topic (optional).

class pyryver.objects.**TopicReply**(*ryver: pyryver.ryver.Ryver, data: dict*)

Bases: `pyryver.objects.PostedComment`

A reply on a topic.

await get_topic() → *pyryver.objects.Topic*

Get the topic this reply was sent to.

Returns The topic associated with this reply.

2.3.5 Tasks

class pyryver.objects.**TaskBoard**(*ryver: pyryver.ryver.Ryver, data: dict*)

Bases: *pyryver.objects.Object*

A Ryver task board.

Variables

- ***BOARD_TYPE_BOARD*** – A task board with categories.
- ***BOARD_TYPE_LIST*** – A task list (i.e. a task board without categories).

BOARD_TYPE_BOARD = 'board'

BOARD_TYPE_LIST = 'list'

get_name() → str

Get the name of this task board.

This will be the same as the name of the forum/team/user the task board is associated with.

Returns The name of the task board.

get_board_type() → str

Get the type of this task board.

Returns one of the `BOARD_TYPE_` constants in this class.

`BOARD_TYPE_BOARD` is a task board with categories, while `BOARD_TYPE_LIST` is a task list without categories.

Not to be confused with *`Object.get_type()`*.

Returns The type of this task board.

get_prefix() → str

Get the prefix for this task board.

The prefix can be used to reference tasks across Ryver using the #PREFIX-ID syntax.

If a task board does not have task IDs set up, this will return None.

Returns The task prefix for this task board.

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

Get all the categories in this task board.

Even if this task board has no categories (a list), this method will still return a single category, "Uncategorized".

Parameters

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

Returns An async iterator for the categories in this task board.

await create_category(*name: str*, *done: bool = False*) → *pyryver.objects.TaskCategory*

Create a new task category in this task board.

Parameters

- **name** – The name of this category.
- **done** – Whether tasks moved to this category should be marked as done.

Returns The created category.

async for ... in get_tasks(*archived: Optional[bool] = None*, *top: int = - 1*, *skip: int = 0*) → AsyncIterator[*pyryver.objects.Task*]

Get all the tasks in this task board.

If `archived` is unspecified or None, all tasks will be retrieved. If `archived` is either True or False, only tasks that are archived or unarchived are retrieved, respectively.

This will not retrieve sub-tasks (checklist items).

Parameters

- **archived** – If True or False, only retrieve tasks that are archived or unarchived; if None, retrieve all tasks (optional).
- **top** – Maximum number of results; optional, if unspecified return all results.
- **skip** – Skip this many results (optional).

Returns An async iterator for the tasks in this task board.

await create_task(*subject: str, body: str = '', category: Optional[TaskCategory] = None, assignees: Optional[Iterable[pyryver.objects.User]] = None, due_date: Optional[str] = None, tags: Union[List[str], List[pyryver.objects.TaskTag], None] = None, checklist: Optional[Iterable[str]] = None, attachments: Optional[Iterable[Union[Storage, File]]] = None*) → *pyryver.objects.Task*

Create a task in this task board.

If the category is None, this task will be put in the "Uncategorized" category. For list type task boards, the category can be left as None.

Tip: To attach files to the task, use `pyryver.ryver.Ryver.upload_file()` to upload the files you wish to attach. Alternatively, use `pyryver.ryver.Ryver.create_link()` for link attachments.

Tip: You can use `pyryver.util.datetime_to_iso8601()` to turn datetime objects into timestamps that Ryver will accept.

Note that timezone info **must** be present in the timestamp. Otherwise, this will result in a 400 Bad Request.

Parameters

- **subject** – The subject, or title of the task.
- **body** – The body, or description of the task (optional).
- **category** – The category of the task; if None, the task will be uncategorized (optional).
- **assignees** – A list of users to assign for this task (optional).
- **due_date** – The due date, as an ISO 8601 formatted string **with a timezone offset** (optional).
- **tags** – A list of tags of this task (optional). Can either be a list of strings or a list of ``TaskTag``s.
- **checklist** – A list of strings which are used as the item names for the checklist of this task (optional).
- **attachments** – A list of attachments for this task (optional).

Returns The created task.

await get_chat() → *pyryver.objects.Chat*

Get the chat this task board is in.

If this task board is a public task board in a forum or team, a `GroupChat` object will be returned. If this task board is a personal (user) task board, a `User` object will be returned.

Note: API Detail: Although public task boards can be in either a forum or a team, `GroupChat` objects returned by this method will *always* be instances of `Team`, even if the task board exists in a forum. There seems to be no way of determining whether the returned chat is actually a forum or a team. However, as both forums/teams have the exact same methods, this detail shouldn't matter.

Returns The forum/team/user this task board is in.

class pyryver.objects.**TaskCategory**(*ryver: pyryver.ryver.Ryver*, *data: dict*)

Bases: *pyryver.objects.Object*

A category in a task board.

Variables

- ***CATEGORY_TYPE_UNCATEGORIZED*** – The "Uncategorized" category, created by the system (present in all task boards regardless of whether it is shown).
- ***CATEGORY_TYPE_DONE*** – A user-created category in which all tasks are marked as done.
- ***CATEGORY_TYPE_OTHER*** – Other categories (user-created and not marked as done).

CATEGORY_TYPE_UNCATEGORIZED = 'uncategorized'

CATEGORY_TYPE_DONE = 'done'

CATEGORY_TYPE_OTHER = 'user'

get_name() → str

Get the name of this category.

Returns The name of this category.

get_position() → int

Get the position of this category in this task board.

Positions are numbered from left to right.

Note: The first user-created category that is shown in the UI has a position of 1. This is because the "Uncategorized" category, which is present in all task boards, always has a position of 0, even when it's not shown (when there are no uncategorized tasks).

Returns The position of this category.

get_category_type() → str

Get the type of this task category.

Returns one of the `CATEGORY_TYPE_` constants in this class.

Returns The type of this category.

await get_task_board() → *pyryver.objects.TaskBoard*

Get the task board that contains this category.

Returns The task board.

await edit(*name: Optional[str] = NO_CHANGE*, *done: Optional[bool] = NO_CHANGE*) → None

Edit this category.

Note: This method also updates these properties in this object.

Warning: `done` should **never** be set for the "Uncategorized" category, as its type cannot be modified. If set, a `ValueError` will be raised.

If any parameters are unspecified or `NO_CHANGE`, they will be left as-is. Passing `None` for parameters for which `None` is not a valid value will also result in the value being unchanged.

Parameters

- **`name`** – The name of this category (optional).
- **`done`** – Whether tasks moved to this category should be marked as done (optional).

Raises `ValueError` – If attempting to modify the type of the "Uncategorized" category.

await delete(*move_to: Optional[TaskCategory] = None*) → None

Delete this category.

If `move_to` is specified, the tasks that are in this category will be moved into the specified category and not archived. Otherwise, the tasks will be archived.

Parameters **`move_to`** – Another category to move the tasks in this category to (optional).

await archive(*completed_only: bool = False*) → None

Archive either all or only completed tasks in this category.

Deprecated since version 0.4.0: Use *`TaskCategory.archive_tasks()`* instead. The functionality is unchanged but the name is less misleading.

Note: This archives the tasks in this category, not the category itself.

Parameters **`completed_only`** – Whether to only archive completed tasks (optional).

await archive_tasks(*completed_only: bool = False*) → None

Archive either all or only completed tasks in this category.

Parameters **`completed_only`** – Whether to only archive completed tasks (optional).

await move_position(*position: int*) → None

Move this category's display position in the UI.

Note: This also updates the position property of this object.

The first user-created category that is shown in the UI has a position of 1. This is because the "Uncategorized" category, which is present in all task boards, always has a position of 0, even when it's not shown (when there are no uncategorized tasks).

Therefore, no user-created task category can ever be moved to position 0, and the "Uncategorized" category should never be moved.

Parameters **`position`** – The new display position.

await move_tasks(*category: pyryver.objects.TaskCategory, completed_only: bool = False*) → None

Move either all or only completed tasks in this category to another category.

Parameters

- **`category`** – The category to move to.
- **`completed_only`** – Whether to only move completed tasks (optional).

async for ... in get_tasks(*archived: Optional[bool] = None, top: int = - 1, skip: int = 0*) → AsyncIterator[*pyryver.objects.Task*]

Get all the tasks in this category.

If `archived` is unspecified or None, all tasks will be retrieved. If `archived` is either True or False, only tasks that are archived or unarchived are retrieved, respectively.

This will not retrieve sub-tasks (checklist items).

Parameters

- **`archived`** – If True or False, only retrieve tasks that are archived or unarchived; if None, retrieve all tasks (optional).
- **`top`** – Maximum number of results; optional, if unspecified return all results.
- **`skip`** – Skip this many results (optional).

Returns An async iterator for the tasks in this category.

class `pyryver.objects.`**`Task`**(*ryver: pyryver.ryver.Ryver*, *data: dict*)

Bases: `pyryver.objects.PostedMessage`

A Ryver task.

`is_archived``()` → bool

Get whether this task has been archived.

Returns Whether this task has been archived.

`get_subject``()` → str

Get the subject (title) of this task.

Returns The subject of this task.

`get_due_date``()` → Optional[str]

Get the due date as an ISO 8601 timestamp.

If there is no due date, this method will return None.

Tip: You can use *`pyryver.util.iso8601_to_datetime()`* to convert the timestamps returned by this method into a datetime.

Returns The due date of this task.

`get_complete_date``()` → Optional[str]

Get the complete date as an ISO 8601 timestamp.

If the task has not been completed, this method will return None.

Tip: You can use *`pyryver.util.iso8601_to_datetime()`* to convert the timestamps returned by this method into a datetime.

Returns The completion date of this task.

`is_completed``()` → bool

Get whether this task has been completed.

Returns Whether this task has been completed.

`get_short_repr``()` → Optional[str]

Get the short representation of this task.

This is can be used to reference this task across Ryver. It has the form PREFIX-ID, and is also unique across the entire organization.

If the task board does not have prefixes set up, this will return None.

Returns The unique short representation of this task.

get_position() → int

Get the position of this task in its category or the task list.

The first task has a position of 0.

Returns The position of this task in its category.

get_comments_count() → int

Get how many comments this task has received.

Returns The number of comments this task has received.

get_attachments_count() → int

Get how many attachments this task has.

Returns The number of attachments this task has.

get_tags() → List[str]

Get all the tags of this task.

Note: The tags are returned as a list of strings, not a list of ``TaskTag``s.

Returns All the tags of this task, as strings.

await get_task_board() → *pyryver.objects.TaskBoard*

Get the task board this task is in.

Returns The task board containing this task.

await get_task_category() → *pyryver.objects.TaskCategory*

Get the category this task is in.

Returns The category containing this task.

await get_assignees() → List[*pyryver.objects.User*]

Get the assignees of this task.

Returns The assignees of this task,.

await set_complete_date(*time: Optional[str] = ''*) → None

Set the complete date of this task, which also marks whether this task is complete.

An optional completion time can be specified in the form of an ISO 8601 timestamp with a timezone offset. If not specified or an empty string, the current time will be used.

Tip: You can use `pyryver.util.datetime_to_iso8601()` to turn datetime objects into timestamps that Ryver will accept.

Note that timezone info **must** be present in the timestamp. Otherwise, this will result in a 400 Bad Request.

If None is used as the time, in addition to clearing the complete date, this task will also be un-completed.

Note: This also updates the complete date property in this object.

If a time of None is given, this task will be marked as uncomplete.

Parameters **`time`** – The completion time (optional).

`await set_due_date`(*time: Optional[str]*)

Set the due date of this task.

The time must be specified as an ISO 8601 timestamp with a timezone offset. It can also be None, in which case there will be no due date.

Tip: You can use `pyryver.util.datetime_to_iso8601()` to turn datetime objects into timestamps that Ryver will accept.

Note that timezone info **must** be present in the timestamp. Otherwise, this will result in a 400 Bad Request.

Note: This also updates the due date property in this object.

Parameters **`time`** – The new due date.

`await complete`() → None

Mark this task as complete.

`await uncomplete`() → None

Mark this task as uncomplete.

`await archive`(*archived: bool = True*) → None

Archive or un-archive this task.

Parameters **`archived`** – Whether the task should be archived.

`await unarchive`() → None

Un-archive this task.

This is the same as calling `Task.archive()` with False.

`await move`(*category:* pyryver.objects.TaskCategory, *position: int*) → None

Move this task to another category or to a different position in the same category.

Note: This also updates the position property of this object.

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

Get the checklist items of this task (subtasks).

If this task has no checklist, an empty list will be returned.

The checklist items are `Task` objects; complete or uncomplete those objects to change the checklist status.

Parameters

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

Returns An async iterator for the tasks in the checklist of this task.

await get_parent() → Optional[*pyryver.objects.Task*]

Get the parent task of this sub-task (checklist item).

This only works if this task is an item in another task's checklist. Otherwise, this will return None.

Returns The parent of this sub-task (checklist item), or None if this task is not a sub-task.

await add_to_checklist(*items: Iterable[str]*) → None

Add items to this task's checklist.

Parameters `items` – The names of the items to add.

await set_checklist(*items: Iterable[Task]*) → None

Set the contents of this task's checklist.

This will overwrite existing checklist items.

Note: This method should be used for deleting checklist items only. It cannot be used to add new items as the tasks would have to be created first. To add new items, use `Task.add_to_checklist()`.

Parameters `items` – The items in the checklist.

await edit(*subject: Optional[str] = NO_CHANGE, body: Optional[str] = NO_CHANGE, category: Optional[TaskCategory] = NO_CHANGE, assignees: Optional[Iterable[*pyryver.objects.User*]] = NO_CHANGE, due_date: Optional[str] = NO_CHANGE, tags: Union[List[str], List[pyryver.objects.TaskTag], None] = NO_CHANGE, attachments: Optional[Iterable[Union[Storage, File]]] = NO_CHANGE*) → None

Edit this task.

Note: Admins can edit any task regardless of whether they created it.

The file attachments (if specified) will **replace** all existing attachments.

Additionally, this method also updates these properties in this object.

Note: While a value of `None` for the category in `TaskBoard.create_task()` will result in the task being placed in the "Uncategorized" category, `None` is not a valid value for the category in this method, and if used will result in the category being unmodified.

This method does not support editing the checklist. To edit the checklist, use `Task.get_checklist()`, `Task.set_checklist()` and `Task.add_to_checklist()`.

If any parameters are unspecified or `NO_CHANGE`, they will be left as-is. Passing `None` for parameters for which `None` is not a valid value will also result in the value being unchanged.

Tip: To attach files to the task, use `pyryver.ryver.Ryver.upload_file()` to upload the files you wish to attach. Alternatively, use `pyryver.ryver.Ryver.create_link()` for link attachments.

Tip: You can use `pyryver.util.datetime_to_iso8601()` to turn datetime objects into timestamps that Ryver will accept.

Note that timezone info **must** be present in the timestamp. Otherwise, this will result in a 400 Bad Request.

Parameters

- **`subject`** – The subject, or title of the task (optional).
- **`body`** – The body, or description of the task (optional).
- **`category`** – The category of the task; if None, the task will be uncategorized (optional).
- **`assignees`** – A list of users to assign for this task (optional).
- **`due_date`** – The due date, as an ISO 8601 formatted string **with a timezone offset** (optional).
- **`tags`** – A list of tags of this task (optional). Can either be a list of strings or a list of ``TaskTag``s.
- **`attachments`** – A list of attachments for this task (optional).

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

Get all the comments on this task.

Parameters

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

Returns An async iterator for the comments of this task.

`await comment`(*message: str*, *attachments: Optional[Iterable[Union[Storage, File]]] = None*, *creator: Optional[pyryver.objects.Creator] = None*) → pyryver.objects.TaskComment

Comment on this task.

Note: For unknown reasons, overriding the creator does not seem to work for this method.

Tip: To attach files to the comment, use *`pyryver.ryver.Ryver.upload_file()`* to upload the files you wish to attach. Alternatively, use *`pyryver.ryver.Ryver.create_link()`* for link attachments.

Changed in version 0.4.0: Switched the order of attachments and creator for consistency.

Parameters

- **`message`** – The comment's contents.
- **`creator`** – The overridden creator (optional). **Does not work.**
- **`attachments`** – A number of attachments for this comment (optional).

Returns The created comment.

2.3.6 Files

class `pyryver.objects.`**`Storage`**(*ryver:* pyryver.ryver.Ryver, *data:* *dict*)

Bases: *`pyryver.objects.Object`*

Generic storage (message attachments), e.g. files and links.

Variables

- ***`STORAGE_TYPE_FILE`*** – An uploaded file.
- ***`STORAGE_TYPE_LINK`*** – A link.

`STORAGE_TYPE_FILE = 'file'`

`STORAGE_TYPE_LINK = 'link'`

`get_storage_type`() → str

Get the type of this storage object.

Returns one of the `STORAGE_TYPE_` constants in this class.

Not to be confused with *`Object.get_type()`*.

Returns The type of this storage object.

`get_name`() → str

Get the name of this storage object.

Returns The name of this storage object.

`get_size`() → str

Get the size of the object stored.

Returns The size of the thing stored.

`get_content_id`() → int

Get the ID of the **contents** of this storage object.

If a link is stored, then this will return the same value as `get_id()`. If a file is stored, then this will return the ID of the file instead.

Returns The ID of the contents of this storage object.

`get_content_MIME_type`() → str

Get the MIME type of the content.

For links, this will be "application/octet-stream".

Returns The MIME type of the contents of this storage object.

`get_file`() → *pyryver.objects.File*

Get the file stored.

Warning: This method will raise a `KeyError` if the type of this storage is not `TYPE_FILE`!

Returns The file stored.

`get_content_url`() → str

Get the URL of the contents.

If a link is stored, then this will be the URL of the link. If a file is stored, then this will be the URL of the file contents.

Returns The content's URL.

await delete() → None
Delete this storage object and the file it contains if there is one.

await make_avatar_of(*chat: pyryver.objects.Chat*) → None
Make this image an avatar of a chat.

Parameters **chat** – The chat to change the avatar for.

Raises **ValueError** – If the contents of this storage object is not a file.

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

An uploaded file.

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

Returns The title of this file.

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

Returns The name of this file.

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

Returns The size of the file in bytes.

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

Returns The URL of the file.

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

Returns The MIME type of the file.

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

Returns an aiohttp request response to the file URL.

Returns An `aiohttp.ClientResponse` object representing a request response for the file contents.

await download_data() → bytes
Download the file data.

Returns The downloaded file data, as raw bytes.

await delete() → None
Delete this file.

2.3.7 Notifications

class pyryver.objects.**Notification**(*ryver: pyryver.ryver.Ryver*, *data: dict*)

Bases: *pyryver.objects.Object*

A Ryver user notification.

Variables

- ***PREDICATE_MENTION*** – The user was directly mentioned with an @mention.
- ***PREDICATE_GROUP_MENTION*** – The user was mentioned through @team or @here.
- ***PREDICATE_COMMENT*** – A topic was commented on.
- ***PREDICATE_TASK_COMPLETED*** – A task was completed.

PREDICATE_MENTION = 'chat_mention'

PREDICATE_GROUP_MENTION = 'group_mention'

PREDICATE_COMMENT = 'commented_on'

PREDICATE_TASK_COMPLETED = 'completed'

get_predicate() → str

Get the "predicate", or type, of this notification.

This usually returns one of the `PREDICATE_` constants in this class. Note that the list currently provided is not exhaustive; this method may return a value that isn't one of the constants.

Returns The predicate of this notification.

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.

Returns The entity type of this notification's subject.

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.

Returns The ID of this notification's subject.

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.

Returns The subjects of this notification.

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.

Returns The entity type of the object of this notification.

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.

Returns The ID of the object of this notification.

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.

Returns The object of this notification.

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.

Returns The entity type of the via of this notification.

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.

Returns The ID of the via of this notification.

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.

Returns The via of this notification.

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

Returns Whether this notification is new.

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

Returns Whether this notification is unread.

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

Note: This also updates these properties in this object.

2.3.8 Creators

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

name

avatar

to_dict() → dict

Convert this Creator object to a dictionary to be used in a request.

> **Warning:** This method is intended for internal use only.

Returns The dict representation of this object.

2.4 Utilities

2.4.1 Cache Data Storage

Cache storages are used by `Ryver` to cache organization data locally.

In large organizations with lots of data, caching can be used to make the program load some organization data locally instead of fetching them from Ryver. This can significantly improve program startup times.

Currently, the lists of all users, forums, and teams can be cached.

See also:

The `Ryver` class

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.

Parameters

- **ryver** – The Ryver session to associate the objects with.
- **obj_type** – The type of the objects to load.

Returns A list of saved objects of that type.

abstractmethod save(*obj_type: str*, *data: Iterable[pyryver.objects.Object]*) → None

Save all objects of a specific type.

Parameters

- **`obj_type`** – The type of the objects to save.
- **`data`** – The objects to save.

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.

Parameters

- **`ryver`** – The Ryver session to associate the objects with.
- **`obj_type`** – The type of the objects to load.

Returns A list of saved objects of that type.

`save`(*obj_type: str*, *data: Iterable[*pyryver.objects.Object*]*) → None

Save all objects of a specific type.

Parameters

- **`obj_type`** – The type of the objects to save.
- **`data`** – The objects to save.

2.4.2 API Helpers

This module contains various contants and utilities for both internal and external use.

`pyryver.util.`**`NO_CHANGE = NO_CHANGE`**

This constant is used in the various `edit()` methods. It's used to indicate that there should be no change to the value of a field, in the cases where `None` is a valid value.

`pyryver.util.`**`get_type_from_entity`**(*entity_type: str*) → Optional[str]

Gets the object type from the entity type.

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

> **Warning:** This function is intended for internal use only.

Parameters **`entity_type`** – The entity type of the object.

Returns The regular type of the object, or None if an invalid type.

await `pyryver.util.`**`retry_until_available`**(*action: Callable[[...], Awaitable[T]]*, **args*, *timeout: Optional[float] = None*, *retry_delay: float = 0.5*, ***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 `retry_delay` 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.

For example, this snippet will try to get a message from a chat by ID with a timeout of 5 seconds, retrying after 1 second if a 404 occurs:

```
message = await pyryver.retry_until_available(chat.get_message, message_id,␣
↪timeout=5.0, retry_delay=1.0)
```

Note: Do not "call" the coro first and pass a future to this function; instead, pass a reference to the coro directly, as seen in the example. This is done because a single future cannot be awaited multiple times, so a new one is created each time the function retries.

Parameters

- **`action`** – The coroutine to run.
- **`timeout`** – The timeout in seconds, or None for no timeout (optional).
- **`retry_delay`** – The duration in seconds to wait before trying again (optional).

Returns The return value of the coroutine.

`pyryver.util.`**`iso8601_to_datetime`**(*timestamp: str*) → datetime.datetime

Convert an ISO 8601 timestamp as returned by the Ryver API into a datetime.

Warning: This function does not handle *all* valid ISO 8601 timestamps; it only tries to handle the ones returned by the Ryver API. It uses the simple format string `"%Y-%m-%dT%H:%M:%S%z"` to parse the timestamp.

Therefore, this function should **not** be used for parsing any ISO timestamp; to do that, consider using `dateutil.parser`, or some alternative method.

Parameters **`timestamp`** – The ISO 8601 timestamp.

`pyryver.util.`**`datetime_to_iso8601`**(*timestamp: datetime.datetime*) → str

Convert a datetime into an ISO 8601 timestamp as used by the Ryver API.

Parameters **`timestamp`** – The datetime to convert.

`pyryver.objects.`**`get_obj_by_field`**(*objs: Iterable[pyryver.objects.Object], field: str, value: Any, case_sensitive: str = True*) → Optional[*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.
- **`case_sensitive`** – Whether the search should be case-sensitive. Can be useful for fields such as username or nickname, which are case-insensitive. Defaults to True. If the field value is not a string, it will be ignored.

Returns The object with the matching field, or None if not found.

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

INDEX