DPY Anti-Spam

Oct 14, 2021

Primary Interface:

DPY Anti-Spam supports discord.py and all forks out of the box assuming they use the `discord` namespace.

If you want to use this with hikari, please enable it by passing `is_using_hikari=True` to the `AntiSpamHandler` constructor.

The package features some built in punishments, these are:

- Per member spam is treated as warns, then kicks followed by bans.
- Per channel spam starts off as a kick straight away followed by bans

CHAPTER 1

Main Interface

This file deals with the AntiSpamHandler as it is the primary Interface for you to interact with.

Note, this is the main entrance to this entire package. As such this should be the only thing you interact with.

Punishment messages won't be sent unless a guild sets a log channel.

This handler propagation method also returns the following class for you to use:

antispam.CorePayload

class antispam.**AntiSpamHandler**(*bot*, ***, *is_using_hikari: bool = False*, *options: antispam.dataclasses.options.Options = None*, *cache: antispam.abc.Cache = None*)

The overall handler for the DPY Anti-spam package

DEFAULTS:

warn_threshold: 3 This is the amount of duplicates that result in a warning within the message_interval

kick_threshold: 2 This is the amount of warns required before a kick is the next punishment

ban_threshold: 2 This is the amount of kicks required before a ban is the next punishment

message_interval: 30000ms (30 Seconds) Amount of time a message is kept before being discarded. Essentially the amount of time (In milliseconds) a message can count towards spam

guild_warn_message: "Hey $MENTIONUSER, please stop spamming/sending duplicate messages." The message to be sent in the guild upon warn_threshold being reached

guild_kick_message: "$USERNAME was kicked for spamming/sending duplicate messages." The message to be sent in the guild upon kick_threshold being reached

guild_ban_message: "$USERNAME was banned for spamming/sending duplicate messages." The message to be sent in the guild upon ban_threshold being reached

member_kick_message ["Hey $MENTIONUSER, you are being kicked from $GUILDNAME for spamming/sending duplicate messages."] The message to be sent to the user who is being warned

member_ban_message ["Hey $MENTIONUSER, you are being banned from $GUILDNAME for spamming/sending duplicate messages."] The message to be sent to the user who is being banned

member_failed_kick_message ["I failed to punish you because I lack permissions, but still you shouldn't spam"] The message to be sent to the user if the bot fails to kick them

member_failed_ban_message ["I failed to punish you because I lack permissions, but still you shouldn't spam"] The message to be sent to the user if the bot fails to ban them

message_duplicate_count: 5 The amount of duplicate messages needed within message_interval to trigger a punishment

message_duplicate_accuracy: 90 How 'close' messages need to be to be registered as duplicates (Out of 100)

delete_spam: False Whether or not to delete messages marked as spam

Won't delete messages if `no_punish` *is* `True`

Note, this method is expensive. It will all messages marked as spam, and this means an api call per message.

mention_on_embed: True If the message your trying to send is an embed, also send some content to mention the person being punished.

ignored_members: [] The users (ID Form), that bypass anti-spam

ignored_channels: [] Channels (ID Form), that bypass anti-spam

ignored_roles: [] The roles (ID Form), that bypass anti-spam

ignored_guilds: [] Guilds (ID Form), that bypass anti-spam

ignore_bots: True Should bots bypass anti-spam?

warn_only: False Whether or not to only warn users, this means it will not kick or ban them

no_punish: False Don't punish anyone, simply return whether or not they should be punished within propagate. This essentially lets the end user handle punishments themselves.

To check if someone should be punished, use the returned value from the `propagate` method. If should_be_punished_this_message is True then this package believes they should be punished. Otherwise just ignore that message since it shouldn't be punished.

per_channel_spam: False Track spam as per channel, rather then per guild.

guild_warn_message_delete_after: None The time to delete the `guild_warn_message` message

user_kick_message_delete_after: None The time to delete the `member_kick_message` message

guild_kick_message_delete_after: None The time to delete the `guild_kick_message` message

user_ban_message_delete_after: None The time to delete the `member_ban_message` message

guild_ban_message_delete_after: None The time to delete the `guild_ban_message` message

delete_zero_width_chars: True Should zero width characters be removed from messages

is_using_hikari: False Set this to True if you are using the package with hikari rather then discord.py

`__init__`(*bot*, ***, *is_using_hikari: bool = False*, *options: antispam.dataclasses.options.Options = None*, *cache: antispam.abc.Cache = None*)

AntiSpamHandler entry point.

Parameters

- **`bot`** – A reference to your discord bot object.
- **`is_using_hikari`** (*`bool`*, *`Optional`*) – Set this to True if you are using this package within hikari rather then discord.py

- **options** (`Options, Optional`) – An instance of your custom Options the handler should use
- **cache** (`Cache, Optional`) – Your choice of backend caching

add_guild_log_channel(*log_channel: int*, *guild_id: int*) → None
Registers a log channel on a guild internally

Parameters

- **log_channel** (*int*) – The channel id you wish to use for logging
- **guild_id** (*int*) – The id of the guild to store this on

Notes

Not setting a log channel means it will not send any punishment messages

add_guild_options(*guild_id: int*, *options: antispam.dataclasses.options.Options*) → None
Set a guild's options to a custom set, rather then the base level set used and defined in ASH initialization

Warning: If using/modifying `AntiSpamHandler.options` to give to this method you will **also** be modifying the overall options.

To get an options item you can modify freely call `AntiSpamHandler.get_options()`, this method will give you an instance of the current options you are free to modify however you like.

Notes

This will override any current settings, if you wish to continue using existing settings and merely change some I suggest using the get_options method first and then giving those values back to this method with the changed arguments

add_ignored_item(*item: int*, *ignore_type: antispam.enums.ignored_types.IgnoreType*) → None
Add an item to the relevant ignore list

Parameters

- **item** (*int*) – The id of the thing to ignore
- **ignore_type** (`IgnoreType`) – An enum representing the item to ignore

Raises `ValueError` – item is not of type int or int convertible

Notes

This will silently ignore any attempts to add an item already added.

clean_cache(*strict=False*) → None
Cleans the internal cache, pruning any old/un-needed entries.

TODO Test these modes Non Strict mode:

- **Member deletion criteria:**
  - warn_count == default
  - kick_count == default

    - duplicate_counter == default
    - duplicate_channel_counter_dict == default
    - addons dict == default
    - Also must have no active messages after cleaning.
- **Guild deletion criteria:**
    - options are not custom
    - log_channel_id is not set
    - addons dict == default
    - Also must have no members stored

Strict mode:

- **Member deletion criteria**
    - Has no active messages
- **Guild deletion criteria**
    - Does not have custom options
    - log_channel_id is not set
    - Has no active members

Parameters **`strict`** (*`bool`*) – Toggles the above

Notes

This is expensive, and likely only required to be run every so often depending on how high traffic your bot is.

`get_guild_options`(*guild_id: int*) → antispam.dataclasses.options.Options

Get the options dataclass for a given guild, if the guild doesnt exist raise an exception

Parameters **`guild_id`** (*`int`*) – The guild to get custom options for

Returns The options for this guild

Return type *Options*

Raises `GuildNotFound` – This guild does not exist

Notes

This returns a copy of the options, if you wish to change the options on the guild you should use the package methods.

`init`() → None

This method provides a means to initialize any async calls cleanly and without asyncio madness.

Notes

This method is guaranteed to be called before the first time propagate runs. However, it will not be run when the class is initialized.

static load_from_dict(*bot*, *data: dict*, ***, *raise_on_exception: bool = True*)
Can be used as an entry point when starting your bot to reload a previous state so you don't lose all of the previous punishment records, etc, etc

Parameters

- **bot** – The bot instance
- **data** (*dict*) – The data to load AntiSpamHandler from
- **raise_on_exception** (*bool*) – Whether or not to raise if an issue is encountered while trying to rebuild `AntiSpamHandler` from a saved state

  If you set this to False, and an exception occurs during the build process. This will return an `AntiSpamHandler` instance **without** any of the saved state and is equivalent to simply doing `AntiSpamHandler(bot)`

Returns A new AntiSpamHandler instance where the state is equal to the provided dict

Return type *AntiSpamHandler*

Warning: Don't provide data that was not given to you outside of the `save_to_dict` method unless you are maintaining the correct format.

Notes

This method does not check for data conformity. Any invalid input will error unless you set `raise_on_exception` to `False` in which case the following occurs

If you set `raise_on_exception` to `False`, and an exception occurs during the build process. This method will return an `AntiSpamHandler` instance **without** any of the saved state and is equivalent to simply doing `AntiSpamHandler(bot)`

propagate(*message*) → Union[antispam.dataclasses.core.CorePayload, dict, None]
This method is the base level intake for messages, then propagating it out to the relevant guild or creating one if that is required

For what this returns please see the top of this page.

Parameters **message** (*Union[discord.Message, hikari.messages.Message]*) – The message that needs to be propagated out

Returns A dictionary of useful information about the Member in question

Return type dict

register_plugin(*plugin*, *force_overwrite=False*) → None
Registers a plugin for usage for within the package

Parameters

- **plugin** – The plugin to register
- **force_overwrite** (*bool*) – Whether to overwrite any duplicates currently stored.

Think of this as calling `unregister_extension` and then proceeding to call this method.

Raises `PluginError` – A plugin with this name is already loaded

Notes

This must be a class instance, and must subclass `BasePlugin`

`remove_guild_log_channel`(*guild_id: int*) → None

Removes a registered guild log channel

Parameters **`guild_id`** (*`int`*) – The guild to remove it from

Notes

Silently ignores guilds which don't exist

`remove_guild_options`(*guild_id: int*) → None

Reset a guilds options to the ASH options

Parameters **`guild_id`** (*`int`*) – The guild to reset

Notes

This method will silently ignore guilds that do not exist, as it is considered to have 'removed' custom options due to how Guild's are created

`remove_ignored_item`(*item: int*, *ignore_type: antispam.enums.ignored_types.IgnoreType*) → None

Remove an item from the relevant ignore list

Parameters

- **`item`** (*`int`*) – The id of the thing to un-ignore
- **`ignore_type`** (`IgnoreType`) – An enum representing the item to ignore

Raises `ValueError` – item is not of type int or int convertible

Notes

This will silently ignore any attempts to remove an item not ignored.

`reset_member_count`(*member_id: int*, *guild_id: int*, *reset_type: antispam.enums.reset_type.ResetType*) → None

Reset an internal counter attached to a User object

Parameters

- **`member_id`** (*`int`*) – The user to reset
- **`guild_id`** (*`int`*) – The guild they are attached to
- **`reset_type`** (`ResetType`) – An enum representing the counter to reset

Notes

Silently ignores if the User or Guild does not exist. This is because in the packages mind, the counts are 'reset' since the default value is the reset value.

`save_to_dict``()` → dict

Creates a 'save point' of the current state for this handler which can then be used to restore state at a later date

Returns The saved state in a dictionary form. You can give this to `load_from_dict` to reload the saved state

Return type dict

Notes

For most expected use-case's the returned `Messages` will be outdated, however, they are included as it is technically part of the current state.

Note that is method is expensive in both time and memory. It has to iterate over every single stored class instance within the library and store it in a dictionary.

For bigger bots, it is likely better you create this process yourself using generators in order to reduce overhead.

Warning: Due to the already expensive nature of this method, all returned option dictionaries are not deepcopied. Modifying them during runtime will cause this library to begin using that modified copy.

`unregister_plugin``(`*plugin_name: str*`)` → None

Used to unregister or remove a plugin that is currently loaded into AntiSpamHandler

Parameters **`plugin_name`** (*`str`*) – The name of the class you want to unregister

Raises `PluginError` – This extension isn't loaded

CHAPTER 2

Cache Choices

Internally all data is 'cached' using an implementation which implements `antispam.abc.Cache`

In the standard package you have the following choices:

- `antispam.caches.MemoryCache` (Default)
- `antispam.caches.RedisCache` (Not yet implemented)

In order to use a cache other then the default one, simply pass in an instance of the cache you wish to use with the `cache` kwarg when initialising your `AntiSpamHandler`.

Here is an example, note `RedisCache` will likely need arguments to init.

```
import discord
from discord.ext import commands

from antispam import AntiSpamHandler
from antispam.caches import RedisCache

bot = commands.Bot(command_prefix="!", intents=discord.Intents.all())
bot.handler = AntiSpamHandler(bot, cache=RedisCache())
```

Once a cache is registered like so, there is nothing else you need to do. The package will simply use that caching mechanism.

Also note, AntiSpamHandler will call `antispam.abc.Cache.initialize()` before any cache operations are undertaken.

CHAPTER 3

Example usages

Note, all of these examples are for discord.py. If you would like another library here, let me know.

3.1 Super duper basic bot

```python
import discord
from discord.ext import commands

from antispam import AntiSpamHandler

bot = commands.Bot(command_prefix="!", intents=discord.Intents.all())
bot.handler = AntiSpamHandler(bot)


@bot.event
async def on_ready():
    # On ready, print some details to standard out
    print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")


@bot.event
async def on_message(message):
    await bot.handler.propagate(message)
    await bot.process_commands(message)


if __name__ == "__main__":
    bot.run("Bot Token Here")
```

3.2 Basic Hikari bot

```python
import hikari
from antispam import AntiSpamHandler

bot = hikari.GatewayBot(
    token="..."
)
handler = AntiSpamHandler(bot, is_using_hikari=True)

@bot.listen()
async def ping(event: hikari.GuildMessageCreateEvent) -> None:
    if event.is_bot or not event.content:
        return

    await handler.propagate(event.message)

bot.run()
```

3.3 How to use templating in a string

```python
from discord.ext import commands

from antispam import AntiSpamHandler

bot = commands.Bot(command_prefix="!")
bot.handler = AntiSpamHandler(bot, ban_message="$MENTIONUSER you are hereby banned␣
↪from $GUILDNAME for spam!")

@bot.event
async def on_ready():
    print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")

@bot.event
async def on_message(message):
    await bot.handler.propagate(message)
    await bot.process_commands(message)

if __name__ == "__main__":
    bot.run("Bot Token")
```

3.4 Cog Based Usage

```python
from discord.ext import commands
from antispam import AntiSpamHandler

class AntiSpamCog(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
        self.bot.handler = AntiSpamHandler(self.bot)

```

(continues on next page)

(continued from previous page)

```python
    @commands.Cog.listener()
    async def on_ready(self):
        print("AntiSpamCog is ready!\n-----\n")

    @commands.Cog.listener()
    async def on_message(self, message):
        await self.bot.handler.propagate(message)

def setup(bot):
    bot.add_cog(AntiSpamCog(bot))
```

3.5 How to use templating in embeds

```python
from discord.ext import commands

from antispam import AntiSpamHandler

bot = commands.Bot(command_prefix="!")

warn_embed_dict = {
    "title": "**Dear $USERNAME**",
    "description": "You are being warned for spam, please stop!",
    "timestamp": True,
    "color": 0xFF0000,
    "footer": {"text": "$BOTNAME", "icon_url": "$BOTAVATAR"},
    "author": {"name": "$GUILDNAME", "icon_url": "$GUILDICON"},
    "fields": [
        {"name": "Current warns:", "value": "$WARNCOUNT", "inline": False},
        {"name": "Current kicks:", "value": "$KICKCOUNT", "inline": False},
    ],
}
bot.handler = AntiSpamHandler(bot, guild_warn_message=warn_embed_dict)

@bot.event
async def on_ready():
    print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")

@bot.event
async def on_message(message):
    await bot.handler.propagate(message)
    await bot.process_commands(message)

if __name__ == "__main__":
    bot.run("Bot Token")
```

3.6 Custom Punishments

```python
from discord.ext import commands

from antispam import AntiSpamHandler
from antispam.plugins import AntiSpamTracker
```

(continues on next page)

(continued from previous page)

```python

bot = commands.Bot(command_prefix="!")
bot.handler = AntiSpamHandler(bot, no_punish=True)
bot.tracker = AntiSpamTracker(bot.handler, 3) # 3 Being how many 'punishment requests
↪' before is_spamming returns True
bot.handler.register_extension(bot.tracker)


@bot.event
async def on_ready():
    # On ready, print some details to standard out
    print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")


@bot.event
async def on_message(message):
    await bot.handler.propagate(message)

    if bot.tracker.is_spamming(message):
        # Insert code to mute the user

        # Insert code to tell admins

        # ETC
        bot.tracker.remove_punishments(message)

    await bot.process_commands(message)

if __name__ == "__main__":
    bot.run("Bot Token")
```

CHAPTER 4

Package Logging

This package features a fairly decent set of built-in logging, the recommend logging level is logging.WARNING or logging.INFO

4.1 Basic Usage

Add this into your main.py/bot.py file, be aware this will also setup logging for discord.py and any other modules which use it.

```python
logging.basicConfig(
    format="%(levelname)s | %(asctime)s | %(module)s | %(message)s",
    datefmt="%d/%m/%Y %I:%M:%S %p",
    level=logging.INFO,
)
```

A more full example,

```python
import logging

import discord
from discord.ext import commands

from antispam import AntiSpamHandler
from jsonLoader import read_json

logging.basicConfig(
    format="%(levelname)s | %(asctime)s | %(module)s | %(message)s",
    datefmt="%d/%m/%Y %I:%M:%S %p",
    level=logging.INFO,
)

bot = commands.Bot(command_prefix="!", intents=discord.Intents.all())

```

(continues on next page)

(continued from previous page)

```python
file = read_json("token")

# Generally you only need/want AntiSpamHandler(bot)
bot.handler = AntiSpamHandler(bot, ignore_bots=False)


@bot.event
async def on_ready():
    # On ready, print some details to standard out
    print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")


@bot.event
async def on_message(message):
    await bot.handler.propagate(message)
    await bot.process_commands(message)


if __name__ == "__main__":
    bot.run(file["token"])
```

CHAPTER 5

Message Templating

This package utilises safe conversions for message arguments within strings.

These use discord.py terms. But the package will work with the library you are using seamlessly. Don't worry about not seeing exact matches.

5.1 Templating Options

The following are all the options you as the user have:

- **MENTIONMEMBER** - This will attempt to mention the user, uses `discord.Member.mention`
- **MEMBERNAME** - This will attempt to state the user's name, uses `discord.Member.display_name`
- **MEMBERID** - This will attempt to state the user's id, uses `discord.Member.id`
- **$BOTNAME** - This will attempt to state your bots name, uses `discord.Guild.me.name`
- **$BOTID** - This will attempt to state your bots id, uses `discord.Guild.me.id`
- **$GUILDNAME** - This will attempt to state the guild's name, uses `discord.Guild.name`
- **$GUILDID** - This will attempt to state the guild's id, uses `discord.Guild.id`
- **$TIMESTAMPNOW** - This exact time formatted as hh:mm:ss AM/PM, dd/mm/yyyy, uses `datetime.datetime.now()`
- **$TIMESTAMPTODAY** - Today's date formatted as dd/mm/yyyy, uses `datetime.datetime.now()`
- **$WARNCOUNT** - How many times the user has been warned so far, uses `AntiSpam.User.warn_count`
- **$KICKCOUNT** - How many times the user has been removed from the guild so far, uses `AntiSpam.User.kick_count`

The following are special case's for embeds:

- **MEMBERAVATAR** - This will attempt to display the user's avatar, uses `discord.Member.avatar_url`
- **$BOTAVATAR** - This will attempt to display the bots avatar, uses `discord.Guild.me.avatar_url`

- **$GUILDICON** - This will attempt to display the guilds icon, uses `discord.Guild.icon_url`

Note: Example usages not final. Usage works in discord.py 1.x.x and 2.x.x + hikari

The above are valid in the following uses:

1. `discord.Embed.set_author(url="")`
2. `discord.Embed.set_footer(icon_url="")`

There are currently no plans to support either `discord.Embed.image` *or* `discord.Embed.thumbnail`

5.2 Templating Usage

You can include the above options in the following arguments when you initialize the package:

- **guild_warn_message**
- **guild_kick_message**
- **guild_ban_message**
- **user_kick_message**
- **user_ban_message**

5.3 Embed Templating

The above options can also be used within embeds, these also support templating with the options defined above. These options are available in the following fields:

1. **title**, `discord.Embed.title`
2. **description**, `discord.Embed.description`
3. **author** -> **name** in `discord.Embed.set_author(name="")`
4. **footer** -> **text** in `discord.Embed.set_footer(text="")`
5. **name** & **value** in `discord.Embed.add_field(name="", value="")`

NOTE: You can add the timestamp field also. Provided it exists, it will be replaced with `discord.Message.created_at` *, no value required.*

CHAPTER 6

Migrating to 1.0

The biggest change from 0.x.x to 1.x.x is that every is now more sanely named in regard to pep8.

Likely missing things here, if you'd like support join our discord and we'd be happy to assist.

6.1 Changes

- Extensions are now called plugins
- `AntiSpamHandler` now takes an `Options` class rather then kwargs to set options.
- *user_* -> *member_*
- When failing to send a message, it now sends it to the guild log channels
- Some type param's are now enums. See `IgnoreType` and `ResetType`
- **:py:method:'AntiSpamHandler.propagate'** now returns `CorePayload` instead of a dict
- Some misc methods on the handler have signature changes
- Package is typed more, however not fully. This is still a work in progress
- Misc changes, no doubt I've missed heaps

6.2 Features

- Added support for *Hikari* and all *discord.py* forks
- **Added a guild log channel setting**
  - *guild_* messages will be sent here if set, otherwise same as before
  - **:py:method:'AntiSpamHandler.add_guild_log_channel'**
  - **:py:method:'AntiSpamHandler.remove_guild_log_channel'**

- Abstracted logic and data storage to be separate. This means you can setup your own cache such as redis. See `Cache`
- Now features an easy way to clean up your cache. See **:py:method:‘AntiSpamHandler.clean_cache‘**
- **New plugins:**
  - `AntiMassMention` - To stop people spam pinging
  - `Stats` - For general package stats
  - `AdminLogs` - An easy way to get evidence on punishments
- Plugins now have direct access to storage within the cache. You should be interacting with `PluginCache` for this.
- Plugins now support blacklisting to stop runs on certain guilds. See Plugin Blacklisting under `Package Plugin System`
- Roughly `150%` faster then 0.x.x on small test cases
- Fully tested, no more pesky regression bugs
- Further documented
- More comprehensive logging, this is greatly improved compared to 0.x.x

6.3 Fixes

- When the package attempts to delete spam messages, it will now actually delete *all* messages marked as spam rather then just the last one.
- Logging now lazily computes variables, this should be a decent speedup

CHAPTER 7

Enum Reference

class antispam.enums.**IgnoreType**

This enum should be using with the following methods:

- *antispam.AntiSpamHandler.add_ignored_item()*
- *antispam.AntiSpamHandler.remove_ignored_item()*

It is used to signify the type of item you wish to ignore within any following propagate calls.

CHANNEL = 1

GUILD = 2

MEMBER = 0

ROLE = 3

class antispam.enums.**ResetType**

This enum should be using with the following methods:

- *antispam.AntiSpamHandler.reset_member_count()*

It is used to signify the type of reset you wish to apply to the given member.

KICK_COUNTER = 1

WARN_COUNTER = 0

CHAPTER 8

Option's Reference

This class represents the Options for both Guilds and the AntiSpamHandler itself. It is important to become familiar with this dataclass.

Options can be set in two ways:

- Set when creating a new object `Options(no_punish=True)`
- Set using an existing object `Options.no_punish = True`

For more in depth meanings and explanations, please see the primary docstring of *`antispam.AntiSpamHandler`*

class antispam.dataclasses.options.**Options**(*, warn_threshold: int = 3, kick_threshold: int = 2, ban_threshold: int = 2, message_interval: int = 30000, message_duplicate_count: int = 5, message_duplicate_accuracy: int = 90, guild_ban_message_delete_after: int = None, guild_kick_message_delete_after: int = None, member_ban_message_delete_after: int = None, guild_warn_message_delete_after: int = None, member_kick_message_delete_after: int = None, guild_warn_message: Union[str, dict] = '$MEMBERNAME was warned for spamming/sending duplicate messages.', guild_kick_message: Union[str, dict] = '$MEMBERNAME was kicked for spamming/sending duplicate messages.', guild_ban_message: Union[str, dict] = '$MEMBERNAME was banned for spamming/sending duplicate messages.', member_warn_message: Union[str, dict] = 'Hey $MENTIONMEMBER, please stop spamming/sending duplicate messages.', member_kick_message: Union[str, dict] = 'Hey $MENTIONMEMBER, you are being kicked from $GUILDNAME for spamming/sending duplicate messages.', member_ban_message: Union[str, dict] = 'Hey $MENTIONMEMBER, you are being banned from $GUILDNAME for spamming/sending duplicate messages.', member_failed_kick_message: Union[str, dict] = "I failed to punish you because I lack permissions, but still you shouldn't spam.", member_failed_ban_message: Union[str, dict] = "I failed to punish you because I lack permissions, but still you shouldn't spam.", ignored_members: Set[int] = NOTHING, ignored_channels: Set[int] = NOTHING, ignored_roles: Set[int] = NOTHING, ignored_guilds: Set[int] = NOTHING, delete_spam: bool = False, ignore_bots: bool = True, warn_only: bool = False, no_punish: bool = False, men-

Options for the AntiSpamHandler, see `AntiSpamHandler` for explanations

`__init__`(**, warn_threshold: int = 3, kick_threshold: int = 2, ban_threshold: int = 2, message_interval: int = 30000, message_duplicate_count: int = 5, message_duplicate_accuracy: int = 90, guild_ban_message_delete_after: int = None, guild_kick_message_delete_after: int = None, member_ban_message_delete_after: int = None, guild_warn_message_delete_after: int = None, member_kick_message_delete_after: int = None, guild_warn_message: Union[str, dict] = '$MEMBERNAME was warned for spamming/sending duplicate messages.', guild_kick_message: Union[str, dict] = '$MEMBERNAME was kicked for spamming/sending duplicate messages.', guild_ban_message: Union[str, dict] = '$MEMBERNAME was banned for spamming/sending duplicate messages.', member_warn_message: Union[str, dict] = 'Hey $MENTIONMEMBER, please stop spamming/sending duplicate messages.', member_kick_message: Union[str, dict] = 'Hey $MENTIONMEMBER, you are being kicked from $GUILDNAME for spamming/sending duplicate messages.', member_ban_message: Union[str, dict] = 'Hey $MENTIONMEMBER, you are being banned from $GUILDNAME for spamming/sending duplicate messages.', member_failed_kick_message: Union[str, dict] = "I failed to punish you because I lack permissions, but still you shouldn't spam.", member_failed_ban_message: Union[str, dict] = "I failed to punish you because I lack permissions, but still you shouldn't spam.", ignored_members: Set[int] = NOTHING, ignored_channels: Set[int] = NOTHING, ignored_roles: Set[int] = NOTHING, ignored_guilds: Set[int] = NOTHING, delete_spam: bool = False, ignore_bots: bool = True, warn_only: bool = False, no_punish: bool = False, mention_on_embed: bool = True, delete_zero_width_chars: bool = True, per_channel_spam: bool = False, is_per_channel_per_guild: bool = True, addons: Dict[str, Any] = NOTHING*) → None

Method generated by attrs for class Options.

`addons`

`ban_threshold`

`delete_spam`

`delete_zero_width_chars`

`guild_ban_message`

`guild_ban_message_delete_after`

`guild_kick_message`

`guild_kick_message_delete_after`

`guild_warn_message`

`guild_warn_message_delete_after`

`ignore_bots`

`ignored_channels`

`ignored_guilds`

`ignored_members`

`ignored_roles`

`is_per_channel_per_guild`

`kick_threshold`

`member_ban_message`

`member_ban_message_delete_after`

`member_failed_ban_message`

`member_failed_kick_message`

`member_kick_message`

`member_kick_message_delete_after`

`member_warn_message`

`mention_on_embed`

`message_duplicate_accuracy`

`message_duplicate_count`

`message_interval`

`no_punish`

`per_channel_spam`

`warn_only`

`warn_threshold`

CHAPTER 9

CorePayload Reference

You should not be creating this object yourself.

class antispam.**CorePayload**(*member_warn_count: int = 0, member_kick_count: int = 0, member_duplicate_count: int = 0, member_status: str = 'Unknown', member_was_warned: bool = False, member_was_kicked: bool = False, member_was_banned: bool = False, member_should_be_punished_this_message: bool = False, pre_invoke_extensions: Dict[str, Any] = NOTHING, after_invoke_extensions: Dict[str, Any] = NOTHING*)

The CorePayload is a dataclasses which gets returned within the core punishment system for this package.

This is returned from the `antispam.AntiSpamHandler.propagate()` method.

Parameters

- **member_warn_count** (*int*) – How many warns this member has at this point in time
- **member_kick_count** (*int*) – How many kicks this member has at this point in time
- **member_duplicate_count** (*int*) – How many messages this member has marked as duplicates
- **member_status** (*str*) – The status of punishment towards the member
- **member_was_warned** (*bool*) – If the default punishment handler warned this member
- **member_was_kicked** (*bool*) – If the default punishment handler kicked this member
- **member_was_banned** (*bool*) – If the default punishment handler banned this member
- **member_should_be_punished_this_message** (*bool*) – If AntiSpamHandler thinks this member should receive some form of punishment this message. Useful for `antispam.plugins.AntiSpamTracker`

__init__(*member_warn_count: int = 0, member_kick_count: int = 0, member_duplicate_count: int = 0, member_status: str = 'Unknown', member_was_warned: bool = False, member_was_kicked: bool = False, member_was_banned: bool = False, member_should_be_punished_this_message: bool = False, pre_invoke_extensions: Dict[str, Any] = NOTHING, after_invoke_extensions: Dict[str, Any] = NOTHING*) → None

Method generated by attrs for class CorePayload.

CHAPTER 10

Package Plugin System

This package features feature a built in plugins framework soon. This framework can be used to hook into the `propagate` method and run as either a **pre_invoke** or **after_invoke** (Where **invoke** is the built in **propagate**)

All registered extensions **must** subclass `BasePlugin`

A plugin can do anything, from AntiProfanity to AntiInvite. Assuming it is class based and follows the required schema you can easily develop your own plugin that can be run whenever the end developer calls `await AntiSpamHandler.propagate()`

Some plugins don't need to be registered as an extension. A good example of this is the `AntiSpamTracker` class. This class does not need to be invoked with `propagate` as it can be handled by the end developer for finer control. However, it can also be used as a plugin if users are happy with the default behaviour.

Any plugin distributed under the antispam package needs to be lib agnostic, so as to not a dependency of something not in use.

10.1 Plugin Blacklisting

Plugins provide a simplistic interface for skipping execution in any given guild. Simply add the guilds id to the set located under the *Plugin.blacklisted_guilds* variable and then this plugin will not be called for said guild.

10.2 Custom Punishments

```python
from discord.ext import commands

from antispam import AntiSpamHandler
from antispam.ext import Stats

bot = commands.Bot(command_prefix="!")
bot.handler = AntiSpamHandler(bot, no_punish=True)
```

(continues on next page)

(continued from previous page)

```python
bot.stats = Stats(bot.handler)
bot.handler.register_extension(bot.stats)

# We don't want to collect stats on guild 12345
# So lets ignore it on this plugin
bot.stats.blacklisted_guilds.add(12345)


@bot.event
async def on_ready():
    # On ready, print some details to standard out
    print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")


if __name__ == "__main__":
    bot.run("Bot Token")
```

10.3 Call Stack

- **Initially all checks are run, these are the checks baked into `AntiSpamHandler`**
  - You cannot avoid these checks, if you wish to mitigate them you should set them to values that will not be triggered
  - An option to run code before checks may be added in a future version, if this is something you would like, jump into discord and let me know! If I know people want features, they get done quicker
- **Following that, all pre-invoke plugins will be run**
  - If the guild this was called on is within *Plugin.blacklisted_guilds* then execution will be skipped and we move onto the next plugin.
  - The ordered that these are run is loosely based on the order that plugins were registered. Do not expect any form of runtime ordering however. You should build them around the idea that they are guaranteed to run before `AntiSpamHandler.propagate`, not other plugins.
  - Returning `cancel_next_invocation: True` will result in `propagate` returning straight away. It will then return the dictionary of currently processed *pre_invoke_extensions*
- **Run `AntiSpamHandler.propagate`**
  - If any pre-invoke plugin has returned a True value for `cancel_next_invocation` then this method, and any after_invoke extensions will not be called.
- **Run all after-invoke plugins**
  - If the guild this was called on is within *Plugin.blacklisted_guilds* then execution will be skipped and we move onto the next plugin.
  - After_invoke plugins get output from both `AntiSpamHandler` and all pre-invoke plugins as a method argument

CHAPTER 11

Plugin Class Schema

All plugins that aim to be used as a registered extension *within* `AntiSpamHandler` should have at least the following class layout.

All registered plugins **must** subclass `BasePlugin`

11.1 Pre-invoke Schema

```python
from antispam import BasePlugin

class Placeholder(BasePlugin):
    def __init__(self):
        self.is_pre_invoke = True

    async def propagate(self, message: discord.Message) -> dict:
        # Do your code stuff here
```

`self.is_pre_invoke` is optional assuming your extension is using a pre-invoke due to the nature of the implementation.

11.2 After-invoke Schema

```python
from antispam import BasePlugin

class Placeholder(BasePlugin):
    def __init__(self):
        self.is_pre_invoke = False

    async def propagate(self, message: discord.Message, propagate_data: CorePayload) -
↪> dict:
        # Do your code stuff here
```

The only difference between these two schema's, outside of `self.is_pre_invoke` being different, is that the after-invoke method will also be given an extra argument which is the data returned by `propagate`

11.3 Cancelling Invocation

If a key called `cancel_next_invocation` is `True` within the return data from any extension, `AntiSpamHandler.propagate` will immediately return without executing any remaining extensions or even `AntiSpamHandler.propagate`

Example usage: Say you want to use AntiSpamHandler, but only if the message doesnt contain a secret word. You would create a pre-invoke extension, and if the secret word is said you would set `cancel_next_invocation` to `True` and then `AntiSpamHandler` would ignore that message. Thats quite cool aint it! Woop woop

CHAPTER 12

Plugin-Cache Interaction

The interface you should use to have your plugin store data in the global cache.

class antispam.**PluginCache**(*handler: antispam.anti_spam_handler.AntiSpamHandler*, *caller*)

This class handles all data storage. You should simply refer to the methods in this class as your means of interacting with the internal cache

__init__(*handler: antispam.anti_spam_handler.AntiSpamHandler*, *caller*)

Parameters

- **handler** (`AntiSpamHandler`) – Your AntiSpamHandler instance
- **caller** (*class*) – *self*, from the class using this class

get_guild_data(*guild_id: int*) → Any

Get a dictionary of all data for this guild that was stored by this class

Parameters **guild_id** (*int*) – The guild to fetch

Returns The data stored on this

Return type Any

Raises `GuildNotFound` – The given guild could not be found in the cache or it has no stored data

get_member_data(*member_id: int*, *guild_id: int*) → Any

Returns a dictionary of data this caller is allowed to access and store how they please

Parameters

- **member_id** (*int*) – The user we want to get data for
- **guild_id** (*int*) – The guild for the user we want

Returns Stored data on this member which has been stored by this class

Return type Any

Raises `MemberNotFound` – The given user/guild could not be found internally or they have no stored data

set_guild_data(*guild_id: int*, *addon_data: Any*) → None

Stores the given addon data dictionary within the guilds cache

Parameters

- **guild_id** (*int*) – The guild to store this on
- **addon_data** (*Any*) – The data to store on this guild

Notes

Silently creates a new Guild as required

set_member_data(*member_id: int*, *guild_id: int*, *addon_data: Any*) → None

Stores a member's data within a guild

Parameters

- **guild_id** (*int*) – The guild to add this user's data into
- **member_id** (*int*) – The user's id to store
- **addon_data** (*Any*) – The data to store

Notes

Silently creates the required Guild / Member objects as needed

CHAPTER 13

AntiSpamTracker Plugin

A cool plugin designed to assist you with custom punishments.

class antispam.plugins.**AntiSpamTracker**(*anti_spam_handler: antispam.anti_spam_handler.AntiSpamHandler*, *spam_amount_to_punish*, *valid_timestamp_interval=None*)

A class devoted to people who want to handle punishments themselves.

This class wraps a few things, and handles the logic of ensuring everything exists (or doesnt) among other things such as untracking users after the valid storage interval expires

In order to use this in your code, you can either:

- Subclass this class and override the `do_punishment` method and then use it that way to keep it clean
- Initialize this class and simply use the bool `is_spamming()` and do punishments based off that
- Initialize this class and simply use `get_user_count()` to get the number of times the user should be punished and do your own logic

This mainly just depends on how granular you want to be within your code base.

The way it works, is everytime you call `propagate` you simply pass the returned data into *update_cache* and it will update said Members cache if AntiSpamHandler thinks that they should be punished. Now, you set `spam_amount_to_punish` when creating an instance of this class and that is used to check if YOU think they should be punished, and what punishment to give when they hit that cap.

Basically:

`propagate` -> `update_cache`, if the User should be punished we increment internal counter

`is_spamming` -> Checks if the User's internal counter meets `spam_amount_to_punish` and returns a bool

__init__(*anti_spam_handler: antispam.anti_spam_handler.AntiSpamHandler*, *spam_amount_to_punish*, *valid_timestamp_interval=None*) → None

Initialize this class and get it ready for usage.

Parameters

- **anti_spam_handler** (AntiSpamHandler) – Your AntiSpamHandler instance
- **spam_amount_to_punish** (*int*) – A number denoting the minimum value required per user in order trip *is_spamming*
- **valid_timestamp_interval** (*int*) – How long a timestamp should remain 'valid' for. Defaults to `AntiSpamHandler.options.get("message_interval")`

  NOTE this is in milliseconds

Raises

- `TypeError` – Invalid Arg Type
- `ValueError` – Invalid Arg Type

anti_spam_handler

do_punishment (*message*, **args*, ***kwargs*) → None
This only exists for if the user wishes to subclass this class and implement there own logic for punishments here.

Parameters message – The message to extract the guild and user from

Notes

This does nothing unless you subclass and implement it yourself.

get_user_count (*message*) → int
Returns how many messages that are still 'valid' (counted as spam) a certain user has

Parameters message – The message from which to extract user

Returns How many times this user has sent a message that has been marked as 'punishment worthy' by AntiSpamHandler within the valid interval time period

Return type int

Raises `MemberNotFound` – The User for the `message` could not be found

is_spamming (*message*) → bool
Given a message, deduce and return if a user is classed as 'spamming' or not based on `punish_min_amount`

Parameters message – The message to extract guild and user from

Returns True if the User is spamming else False

Return type bool

member_tracking

propagate (*message*, *data: Optional[antispam.dataclasses.core.CorePayload] = None*) → dict
Overwrite the base extension to call `update_cache` internally so it can be used as an extension

punish_min_amount

remove_outdated_timestamps (*data: List[T], member_id: int, guild_id: int*) → None
This logic works around checking the current time vs a messages creation time. If the message is older by the config amount it can be cleaned up

Generally not called by the end user

Parameters

- **data** (*List*) – The data to work with

- **`member_id`** (*`int`*) – The id of the member to store on
- **`guild_id`** (*`int`*) – The id of the guild to store on

`remove_punishments`(*message*)

After you punish someone, call this method to 'clean up' there punishments.

Parameters **`message`** – The message to extract user from

Raises `TypeError` – Invalid arg

Notes

This will actually create a member internally if one doesn't already exist for simplicities sake

`update_cache`(*message*, *data: antispam.dataclasses.core.CorePayload*) → None

Takes the data returned from *propagate* and updates this Class's internal cache

Parameters

- **`message`** – The message related to *data's* propagation
- **`data`** (`CorePayload`) – The data returned from *propagate*

`valid_global_interval`

CHAPTER 14

AntiMassMention Plugin

A cool plugin designed to assist you when dealing with mass mentions.

class antispam.plugins.**MassMentionPunishment**(*member_id: int*, *guild_id: int*, *channel_id: int*, *is_overall_punishment: bool*)

This dataclass is what is dispatched when someone should be punished for mention spam.

Parameters

- **member_id** (*int*) – The associated members id
- **channel_id** (*int*) – The associated channels id
- **guild_id** (*int*) – The associated guilds id
- **is_overall_punishment** (*bool*) – If this is `True`, it means the user has exceeded `total_mentions_before_punishment`. Otherwise they have exceeded `min_mentions_per_message`

Notes

You shouldn't be making instances of this.

class antispam.plugins.**AntiMassMention**(*bot*, *handler: antispam.anti_spam_handler.AntiSpamHandler*, ***, *total_mentions_before_punishment: int = 10*, *time_period: int = 15000*, *min_mentions_per_message: int = 5*)

In order to check if you should punish someone, see the below code.

```python
data = await AntiSpamHandler.propagate(message)
return_item: Union[dict, MassMentionPunishment] = data.after_invoke_extensions[
↪"AntiMassMention"]

if isinstance(return_item, MassMentionPunishment):
    # Punish for mention spam
```

`__init__`(*bot*, *handler: antispam.anti_spam_handler.AntiSpamHandler*, ***, *total_mentions_before_punishment: int = 10*, *time_period: int = 15000*, *min_mentions_per_message: int = 5*)

Parameters

- **`bot`** – Our bot instance
- **`handler`** (`AntiSpamHandler`) – Our AntiSpamHandler instance
- **`total_mentions_before_punishment`** (*`int`*) – How many mentions within the time period before we punish the user *Inclusive*
- **`time_period`** (*`int`*) – The time period valid for mentions *Is in milliseconds*
- **`min_mentions_per_message`** (*`int`*) – The minimum amount of mentions in a message before a punishment is issued *Inclusive*

`member = None`

{

"total_mentions": [Tracking(),

]

}

`propagate`(*message*) → Union[dict, antispam.plugins.anti_mass_mention.MassMentionPunishment]

Manages and stores any mass mentions per users

Parameters **`message`** – The message to interact with

Returns

- *dict* – A dictionary explaining what actions have been taken
- *MassMentionPunishment* – Data surrounding the punishment you should be doing.

CHAPTER 15

Statistics Plugin

A simplistic approach to statistics gathering which works by default and requires no further setup.

class antispam.plugins.**Stats**(*anti_spam_handler: antispam.anti_spam_handler.AntiSpamHandler*)

A simplistic approach to aggregating statistics across the anti spam package.

Do note however, it assumes plugins do not error out. If a plugin errors out, this will be inaccurate.

This does play with internals a bit, however, it is distributed within the library I am okay modifying the base package to make this work even better.

__init__(*anti_spam_handler: antispam.anti_spam_handler.AntiSpamHandler*)

Initialize self. See help(type(self)) for accurate signature.

injectable_nonce = 'Issa me, Mario!'

propagate(*message*, *data: antispam.dataclasses.core.CorePayload*) → dict

This method is called whenever the base `antispam.propagate` is called, adhering to `self.is_pre_invoke`

Parameters

- **message** (*Union[discord.Message, hikari.messages.Message]*) – The message to run propagation on
- **data** (*Optional[CorePayload]*) – Optional input given to after invoke plugins which is the return value from the main *propagate()*

Returns A dictionary of useful data to the end user

Return type dict

CHAPTER 16

AdminLogs Plugin

A plugin design to save admins hassle with regard to evidence collection on automated punishments.

Simply register this as a plugin, and it will save the relevant information for all punishments to a text file.

class `antispam.plugins.`**`AdminLogs`**(*handler: antispam.anti_spam_handler.AntiSpamHandler*, *log_location: str*)

A plugin design to save admins hassle with regard to evidence collection on automated punishments.

`__init__`(*handler: antispam.anti_spam_handler.AntiSpamHandler*, *log_location: str*)

Parameters

- **`handler`** (`AntiSpamHandler`) – Our AntiSpamHandler instance
- **`log_location`** – The directory to store logs in, relative from the caller location. This directory should be empty or only contain previous output from this plugin.

Notes

This will save transcripts for *every* punishment, but it only sends ones to discord if the Guild has a log_channel_id set.

`propagate`(*message*, *data: antispam.dataclasses.core.CorePayload = None*) → Any

This method is called whenever the base `antispam.propagate` is called, adhering to `self.is_pre_invoke`

Parameters

- **`message`** (`Union[discord.Message, hikari.messages.Message]`) – The message to run propagation on
- **`data`** (`Optional[CorePayload]`) – Optional input given to after invoke plugins which is the return value from the main *propagate()*

Returns A dictionary of useful data to the end user

Return type dict

CHAPTER 17

Object Overview

The purpose of this section is to inform developers a bit more about how this package works Internally. For the everyday user, this will not be needed. It is aimed at plugin developers who need to interact with the internals.

Anyway, internally within the rewritten package all data is stored within a slotted `attrs` dataclass. Thus was picked over regular class's to stop boiler plate. It is also better for its given use case when compared to a dictionary as it is a fairly set size.

In the initial versions, we also included logic wrapped in the same class but after the move to dataclasses the logic was abstracted out to reduce memory overhead.

17.1 Plugin developers

You shouldn't in most cases be interacting directly with these class's as the package provides an interface for getting and setting data. Your main focus is within the `addons` variable which is a dictionary which maps the name of your plugin class to the data you wish to store.

CHAPTER 18

Abc Reference

This `Protocol` simply defines how a `Cache` should work. This is going to only be useful if you either plan on working directly with an existing cache or wish to build your own.

Any form of internal cache is guranteed to implement this so you can treat it as a source of truth for usage. *(Unless you bypass them)*

class `antispam.abc.`**`Cache`**(**args*, ***kwargs*)

A generic Protocol for any Cache to implement

`add_message`(*message: antispam.dataclasses.message.Message*) → None

Adds a Message to the relevant Member, creating the Guild/Member if they don't exist

Parameters **`message`** (`Message`) – The Message to add to the internal cache

Notes

This should silently create any Guild's/Member's required to fulfil this transaction

`delete_guild`(*guild_id: int*) → None

Removes a guild from the cache.

Parameters **`guild_id`** (*`int`*) – The id of the guild we wish to remove

Notes

This fails silently.

`delete_member`(*member_id: int*, *guild_id: int*) → None

Removes a member from the cache.

Parameters

- **`member_id`** (*`int`*) – The id of the member we wish to remove
- **`guild_id`** (*`int`*) – The guild this member is in

Notes

This fails silently.

drop() → None
Drops the entire cache, deleting everything contained within.

get_all_guilds() → AsyncIterable[antispam.dataclasses.guild.Guild]
Returns a generator containing all cached guilds

Yields *Guild* – A generator of all stored guilds

get_all_members(*guild_id: int*) → AsyncIterable[antispam.dataclasses.member.Member]
Fetches all members within a guild and returns them within a generator

Parameters **guild_id** (`int`) – The guild we want members in

Yields *Member* – All members in the given guild

Raises `GuildNotFound` – The given guild was not found

get_guild(*guild_id: int*) → antispam.dataclasses.guild.Guild
Fetch a Guild dataclass populated with members

Parameters **guild_id** (`int`) – The id of the Guild to retrieve from cache

Raises `GuildNotFound` – A Guild could not be found in the cache with the given id

get_member(*member_id: int*, *guild_id: int*) → antispam.dataclasses.member.Member
Fetch a Member dataclass populated with messages

Parameters

- **member_id** (`int`) – The id of the member to fetch from cache
- **guild_id** (`int`) – The id of the guild this member is associated with

Raises

- `MemberNotFound` – This Member could not be found on the associated Guild within the internal cache
- `GuildNotFound` – The relevant guild could not be found

initialize(**args*, ***kwargs*) → None
This method gets called once when the AntiSpamHandler init() method gets called to allow for setting up connections, etc

Notes

This is not required.

reset_member_count(*member_id: int*, *guild_id: int*, *reset_type: antispam.enums.reset_type.ResetType*) → None
Reset the chosen enum type back to the default value

Parameters

- **member_id** (`int`) – The Member to reset
- **guild_id** (`int`) – The guild this member is in
- **reset_type** (`ResetType`) – An enum denoting the type of reset

set_guild(*guild: antispam.dataclasses.guild.Guild*) → None
Stores a Guild in the cache

This is essentially a UPSERT operation

Parameters **guild** (`Guild`) – The Guild that needs to be stored

set_member(*member: antispam.dataclasses.member.Member*) → None
Stores a Member internally and attaches them to a Guild, creating the Guild silently if required

Essentially an UPSERT operation

Parameters **member** (`Member`) – The Member we want to cache

class antispam.abc.**Lib**(**args*, ***kwargs*)
A protocol to extend and implement for any libs that wish to hook into this package and work natively.

Notes

Not public api. For internal usage only.

check_message_can_be_propagated(*message*) → antispam.dataclasses.propagate_data.PropagateData
Given a message from the relevant package, run all checks to check if this message should be propagated.

Parameters **message** (*Union[discord.Message, hikari.messages.Message]*) – The message to check

Returns The data required within propagate

Return type *PropagateData*

Raises `PropagateFailure` – This raises an error with the *.data* attribute set. *.data* is what get returned from within propagate

create_message(*message*) → antispam.dataclasses.message.Message
Given a message to extract data from, create and return a Message class

Parameters **message** (*Union[discord.Message, hikari.messages.Message]*) – The message to extract data from

Returns The flushed out message

Return type *Message*

delete_member_messages(*member: antispam.dataclasses.member.Member*) → None
Given a member, traverse all duplicate messages and delete them.

Parameters **member** (`Member`) – The member whose messages should be deleted

Notes

Just call delete_message on each message

delete_message(*message*) → None
Given a message, call and handle the relevant deletion contexts.

Parameters **message** (*Union[discord.Message, hikari.messages.Message]*) – The message to delete

Notes

This should handle given errors silently.

dict_to_embed(*data: dict*, *message*, *warn_count: int*, *kick_count: int*)

Parameters

- **data** (*dict*) – The data to build an embed from
- **message** (*Union[discord.Message, hikari.messages.Message]*) – The message to extract data from
- **warn_count** (*int*) – How many warns this person has
- **kick_count** (*int*) – How many kicks this person has

Returns

Return type Union[discord.Embed, hikari.embeds.Embed]

embed_to_string(*embed*) → str

Given an embed, return a string representation

Parameters **embed** (*Union[discord.Embed, hikari.embeds.Embed]*) – The embed to cast to string

Returns The embed as a string

Return type str

get_channel_by_id(*channel_id: int*)

Returns the given channel for the id

get_channel_from_message(*message*)

Returns the channel for a message

get_channel_id(*message*) → int

Returns the channel id of this message

get_file(*path: str*)

Returns a discord file object for the given path

get_guild_id(*message*) → int

Returns the guild id of this message

get_message_mentions(*message*)

Returns all the mentions from a message

punish_member(*original_message*, *member: antispam.dataclasses.member.Member*, *internal_guild: antispam.dataclasses.guild.Guild*, *user_message*, *guild_message*, *is_kick: bool*, *user_delete_after: int = None*, *channel_delete_after: int = None*)

A generic method to handle multiple methods of punishment for a user. Supports: kicking, banning

Parameters

- **member** (Member) – A reference to the member we wish to punish
- **internal_guild** (Guild) – A reference to the guild this member is in
- **original_message** (*Union[discord.Message, hikari.messages.Message]*) – Where we get everything from :)
- **user_message** (*Union[str, discord.Embed, hikari.embeds.Embed]*) – A message to send to the user who is being punished

- **guild_message** (*Union[str, discord.Embed, hikari.embeds.Embed]*) – A message to send in the guild for whoever is being punished
- **is_kick** (*bool*) – Is it a kick? Else ban
- **user_delete_after** (*int, Optional*) – An int value denoting the time to delete user sent messages after
- **channel_delete_after** (*int, Optional*) – An int value denoting the time to delete channel sent messages after

Raises MissingGuildPermissions – I lack perms to carry out this punishment

send_guild_log(*guild, message, delete_after_time: Optional[int], original_channel, file=None*) → None

Sends a message to the guilds log channel

Notes

If no log channel, send in ctx.channel

Parameters

- **guild** (*Guild*) – The guild we wish to send this too
- **message** (*Union[str, discord.Embed, hikari.embeds.Embed]*) – What to send to the guilds log channel
- **delete_after_time** (*Optional[int]*) – How long to delete these messages after
- **original_channel** (*Union[discord.abc.GuildChannel, discord.abc.PrivateChannel, hikari.GuildTextChannel]*) – Where to send the message assuming this guild has no guild log channel already set.
- **file** – A file to send

Notes

This should catch any sending errors, log them and then proceed to return None

send_message_to_(*target, message, mention: str, delete_after_time: Optional[int] = None*) → None

Given a message and target, send :param target: Where to send the message :type target: Union[discord.abc.Messageable, hikari TODO doc this] :param message: The message to send :type message: Union[str, discord.Embed, hikari.embeds.Embed] :param mention: A string denoting a raw mention of the punished user :type mention: str :param delete_after_time: When to delete the message after :type delete_after_time: Optional[int]

Notes

This should implement Options.mention_on_embed

substitute_args(*message: str, original_message, warn_count: int, kick_count: int*) → str

Given a message, substitute in relevant arguments and return a valid string

Parameters

- **message** (*str*) – The message to substitute args into
- **original_message** (*Union[discord.Message, hikari.messages.Message]*) – The message to extract data from

- **`warn_count`** (*`int`*) – How many warns this person has
- **`kick_count`** (*`int`*) – How many kicks this person has

Returns The message with substituted args

Return type str

`transform_message`(*item: Union[str, dict], message, warn_count: int, kick_count: int*)

Parameters

- **`item`** (*`Union[str, dict]`*) – The data to substitute
- **`message`** (*`Union[discord.Message, hikari.messages.Message]`*) – The message to extract data from
- **`warn_count`** (*`int`*) – How many warns this person has
- **`kick_count`** (*`int`*) – How many kicks this person has

Returns

- *Union[str, discord.Embed, hikari.embeds.Embed]*
- *A template complete message ready for sending*

`visualizer`(*content: str*, *message*, *warn_count: int = 1*, *kick_count: int = 2*)

Returns a message transformed as if the handler did it

Parameters

- **`content`** (*`Union[str, discord.Embed, hikari.embeds.Embed]`*) – What to transform
- **`message`** (*`Union[discord.Message, hikari.messages.Message]`*) – Where to extract our values from
- **`warn_count`** (*`int`*) – The warns to visualize with
- **`kick_count`** (*`int`*) – The kicks to visualize with

Returns The transformed content

Return type Union[str, discord.Embed]

CHAPTER 19

ASH Exceptions

Note, these classes should not be used by you. Only use the AntiSpamHandler to work with this package.

All exceptions subclass a base exception `BaseASHException` which provides functionality for error messages

exception `antispam.exceptions.`**`BaseASHException`**(**args*)
A base exception handler for the ASH ecosystem.

`__init__`(**args*)
Initialize self. See help(type(self)) for accurate signature.

exception `antispam.exceptions.`**`DuplicateObject`**(**args*)
Raised because you attempted to create and add an object, using the exact same id's as a pre-existing one.

exception `antispam.exceptions.`**`GuildAddonNotFound`**(**args*)
This class has not addon stored on this guild.

exception `antispam.exceptions.`**`GuildNotFound`**(**args*)
A Guild matching this guild id could not be found in the cache.

exception `antispam.exceptions.`**`InvocationCancelled`**(**args*)
Called when a pre-invoke plugin returned *cancel_next_invocation*

exception `antispam.exceptions.`**`LogicError`**(**args*)

Raised because internal logic has failed. Please create an issue in the github.

exception `antispam.exceptions.`**`MemberAddonNotFound`**(**args*)

This class has not addon stored on this member.

exception `antispam.exceptions.`**`MemberNotFound`**(**args*)

A Member matching this id and guild id could not be found in the cache.

exception `antispam.exceptions.`**`MissingGuildPermissions`**(**args*)

I need both permissions to kick & ban people from this guild in order to work!

exception `antispam.exceptions.`**`NotFound`**(**args*)

Something could not be found.

exception `antispam.exceptions.`**`ObjectMismatch`**(**args*)

Raised because you attempted add a message to a member, but that member didn't create that message.

exception `antispam.exceptions.`**`PluginError`**(**args*)

An error occurred that was related to a plugin and not AntiSpamHandler

exception `antispam.exceptions.`**`PropagateFailure`**(**args*, *data: dict*)

`__init__`(**args*, *data: dict*)

Initialize self. See help(type(self)) for accurate signature.

CHAPTER 20

Guild Reference

You should not be creating this object yourself. It is just useful to understand how they work for say, plugin development.

Internally the guild object wraps a couple layers of data to handle custom options as well as providing an O(1) way of storing Members.

class `antispam.dataclasses.guild.`**`Guild`**(*id: int*, *options: antispam.dataclasses.options.Options = NOTHING*, *log_channel_id: int = None*, *members: Dict[int*, *antispam.dataclasses.member.Member] = NOTHING*, *messages: List[antispam.dataclasses.message.Message] = NOTHING*, *addons: Dict[str*, *Any] = NOTHING*)

A simplistic dataclass representing a Guild

`__init__`(*id: int*, *options: antispam.dataclasses.options.Options = NOTHING*, *log_channel_id: int = None*, *members: Dict[int*, *antispam.dataclasses.member.Member] = NOTHING*, *messages: List[antispam.dataclasses.message.Message] = NOTHING*, *addons: Dict[str*, *Any] = NOTHING*) → None

Method generated by attrs for class Guild.

`addons`

`id`

`log_channel_id`

`members`

`messages`

`options`

CHAPTER 21

Member Reference

You should not be creating this object yourself. It is just useful to understand how they work for say, plugin development.

Internally this object provides an O(1) way of storing Messages as well as maintaining the requried data to track and punish spammers

class antispam.dataclasses.member.**Member**(*id: int, guild_id: int, warn_count: int = 0, kick_count: int = 0, duplicate_counter: int = 1, duplicate_channel_counter_dict: Dict[int, int] = NOTHING, in_guild: bool = True, messages: List[antispam.dataclasses.message.Message] = NOTHING, addons: Dict[str, Any] = NOTHING*)

A simplistic dataclass representing a Member

__init__(*id: int, guild_id: int, warn_count: int = 0, kick_count: int = 0, duplicate_counter: int = 1, duplicate_channel_counter_dict: Dict[int, int] = NOTHING, in_guild: bool = True, messages: List[antispam.dataclasses.message.Message] = NOTHING, addons: Dict[str, Any] = NOTHING*) → None

Method generated by attrs for class Member.

addons

duplicate_channel_counter_dict

duplicate_counter

guild_id

id

kick_count

messages

warn_count

CHAPTER 22

Message Reference

You should not be creating this object yourself. It is just useful to understand how they work for say, plugin development.

Internally the Message object just takes a few attributes from `discord.Message` and stores them in a smaller object to save on memory. It also maintains a `is_duplicate` bool for internal reasons.

class `antispam.dataclasses.message.`**`Message`**(*id: int*, *channel_id: int*, *guild_id: int*, *author_id: int*, *content: str*, *creation_time: datetime.datetime = NOTHING*, *is_duplicate: bool = False*)

A simplistic dataclass representing a Message

`__init__`(*id: int*, *channel_id: int*, *guild_id: int*, *author_id: int*, *content: str*, *creation_time: datetime.datetime = NOTHING*, *is_duplicate: bool = False*) → None

Method generated by attrs for class Message.

`author_id`

`channel_id`

`content`

`creation_time`

`guild_id`

`id`

`is_duplicate`

CHAPTER 23

RedisCache Reference

A caching option within the standard package.

Furthermore, refer to `antispam.abc.Cache` for protocol implementation.

Not yet made

class antispam.caches.**RedisCache**(*handler*)

Not implemented lol

__init__(*handler*)

Initialize self. See help(type(self)) for accurate signature.

CHAPTER 24

MemoryCache Reference

This is the default cache for the package. You shouldn't need to implement it yourself.

Furthermore, refer to `antispam.abc.Cache` for protocol implementation.

class antispam.caches.**MemoryCache**(*handler*)

`__init__`(*handler*)
Initialize self. See help(type(self)) for accurate signature.

`add_message`(*message: antispam.dataclasses.message.Message*) → None
Adds a Message to the relevant Member, creating the Guild/Member if they don't exist

Parameters **`message`** (`Message`) – The Message to add to the internal cache

Notes

This should silently create any Guild's/Member's required to fulfil this transaction

`delete_guild`(*guild_id: int*) → None
Removes a guild from the cache.

Parameters **`guild_id`** (`int`) – The id of the guild we wish to remove

Notes

This fails silently.

`delete_member`(*member_id: int, guild_id: int*) → None
Removes a member from the cache.

Parameters

- **`member_id`** (`int`) – The id of the member we wish to remove
- **`guild_id`** (`int`) – The guild this member is in

Notes

This fails silently.

`drop``()` → None
Drops the entire cache, deleting everything contained within.

`get_all_guilds``()` → AsyncIterable[antispam.dataclasses.guild.Guild]
Returns a generator containing all cached guilds

Yields *Guild* – A generator of all stored guilds

`get_all_members``(`*guild_id: int*`)` → AsyncIterable[antispam.dataclasses.member.Member]
Fetches all members within a guild and returns them within a generator

Parameters **`guild_id`** (`int`) – The guild we want members in

Yields *Member* – All members in the given guild

Raises `GuildNotFound` – The given guild was not found

`get_guild``(`*guild_id: int*`)` → antispam.dataclasses.guild.Guild
Fetch a Guild dataclass populated with members

Parameters **`guild_id`** (`int`) – The id of the Guild to retrieve from cache

Raises `GuildNotFound` – A Guild could not be found in the cache with the given id

`get_member``(`*member_id: int*, *guild_id: int*`)` → antispam.dataclasses.member.Member
Fetch a Member dataclass populated with messages

Parameters

- **`member_id`** (`int`) – The id of the member to fetch from cache
- **`guild_id`** (`int`) – The id of the guild this member is associated with

Raises

- `MemberNotFound` – This Member could not be found on the associated Guild within the internal cache
- `GuildNotFound` – The relevant guild could not be found

`initialize``(`**args*, ***kwargs*`)` → None
This method gets called once when the AntiSpamHandler init() method gets called to allow for setting up connections, etc

Notes

This is not required.

`reset_member_count``(`*member_id: int*, *guild_id: int*, *reset_type: antispam.enums.reset_type.ResetType*`)` → None
Reset the chosen enum type back to the default value

Parameters

- **`member_id`** (`int`) – The Member to reset
- **`guild_id`** (`int`) – The guild this member is in
- **`reset_type`** (`ResetType`) – An enum denoting the type of reset

`set_guild`(*guild: antispam.dataclasses.guild.Guild*) → None
Stores a Guild in the cache

This is essentially a UPSERT operation

Parameters **`guild`** (`Guild`) – The Guild that needs to be stored

`set_member`(*member: antispam.dataclasses.member.Member*) → None
Stores a Member internally and attaches them to a Guild, creating the Guild silently if required

Essentially an UPSERT operation

Parameters **`member`** (`Member`) – The Member we want to cache

CHAPTER 25

PropagateData Object Reference

class antispam.dataclasses.propagate_data.**PropagateData**(*guild_id: int*, *member_name: str*, *member_id: int*, *has_perms_to_make_guild: bool*)

A simplistic dataclass representing the data propagate needs

__init__(*guild_id: int*, *member_name: str*, *member_id: int*, *has_perms_to_make_guild: bool*) → None

Method generated by attrs for class PropagateData.

guild_id

has_perms_to_make_guild

member_id

member_name

CHAPTER 26

Install Notes

Initial install will get you a working version of this lib, however it is recommended you also install **python-Levenshtein** to speed this up. This does require c++ build tools, hence why it is not included by default.

CHAPTER 27

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index