
Discord Anti-Spam

Skelmis

May 12, 2022

PRIMARY INTERFACE:

1	Main Interface	3
2	Cache Choices	9
2.1	Redis Cache	9
2.2	MongoDB Cache	10
3	Possible Modes	11
3.1	Default	11
3.2	Per Member Per Channel	11
3.3	Max Message Amount	11
4	Example usages	13
4.1	Super duper basic bot	13
4.2	Basic Hikari bot	13
4.3	How to use templating in a string	14
4.4	Cog Based Usage	14
4.5	How to use templating in embeds	15
4.6	Custom Punishments	16
5	Package Logging	17
5.1	Basic Usage	17
6	Message Templating	19
6.1	Templating Options	19
6.2	Templating Usage	20
6.3	Embed Templating	20
7	Migrating to 1.0	21
7.1	Changes	21
7.2	Features	21
7.3	Fixes	22
8	Changelog	23
8.1	1.3.0 -> 1.4.0	23
8.2	1.2.11 -> 1.3.0	23
8.3	1.2.8 -> 1.2.9	24
8.4	1.2.(3-7) -> 1.2.8	24
8.5	1.2.2 -> 1.2.3	24
8.6	1.2.1 -> 1.2.2	24
8.7	1.2.0 -> 1.2.1	25
8.8	1.1.3 -> 1.2.0	25

8.9	1.1.2 -> 1.1.3	26
9	Enum Reference	27
10	Option's Reference	29
11	CorePayload Reference	37
12	Package Plugin System	39
12.1	Plugin Blacklisting	39
12.2	Call Stack	39
13	Plugin Class Schema	41
13.1	Pre-invoke Schema	41
13.2	After-invoke Schema	41
13.3	Cancelling Invocation	42
14	Plugin-Cache Interaction	43
15	AntiSpamTracker Plugin	45
16	AntiMassMention Plugin	49
17	Statistics Plugin	51
18	AdminLogs Plugin	53
19	MaxMessageLimiter Plugin	55
20	Object Overview	57
20.1	Plugin developers	57
21	Abc Reference	59
22	ASH Exceptions	69
23	Guild Reference	71
24	Member Reference	73
25	Message Reference	75
26	RedisCache Reference	77
27	MemoryCache Reference	79
28	MongoCache Reference	83
29	PropagateData Object Reference	85
30	Lib Base Object	87
31	SubstituteArgs Object	89
32	Plugin BasePlugin Object	91
33	Install Notes	93

34 Indices and tables	95
Python Module Index	97
Index	99

In order to tell `AntiSpamHandler` what library you are using you should pass the `library` kwarg. For the supported libraries, please see the `Library` enum in the Enum reference page.

The package features some built in punishments, these are:

- **Per member spam:**
 - Warn -> Kick -> Ban. This is default in versions lower then 1.3.0
 - Discord timeouts. This is default in versions 1.3.0 or higher.

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, library: antispam.enums.library.Library, *, options:  
                                Optional[antispam.dataclasses.options.Options] = None, cache:  
                                Optional[antispam.abc.cache.Cache] = None)
```

The overall handler for the DPY Anti-spam package

```
__init__(bot, library: antispam.enums.library.Library, *, options:  
          Optional[antispam.dataclasses.options.Options] = None, cache:  
          Optional[antispam.abc.cache.Cache] = None)
```

AntiSpamHandler entry point.

Parameters

- **bot** – A reference to your discord bot object.
- **library** ([*Library*](#), *Optional*) – An enum denoting the library this AntiSpamHandler. See [*antispam.enums.library.Library*](#) for more
- **options** ([*Options*](#), *Optional*) – An instance of your custom Options the handler should use
- **cache** ([*Cache*](#), *Optional*) – Your choice of backend caching

```
async 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

async `add_guild_options(guild_id: int, options: antis spam.dataclasses.options.Options)` → *None*

Set a guild's options to a custom set, rather than 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 `antis spam.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: antis spam.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.

async `clean_cache(strict=False)` → *None*

Cleans the internal cache, pruning any old/un-needed entries.

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.

async `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.

async `get_options() → antispam.dataclasses.options.Options`

Returns a safe to modify instance of this handlers options.

Returns The safe to use options

Return type *Options*

async `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.

async `static load_from_dict(bot, data: dict, library: antispam.enums.library.Library, *,
raise_on_exception: bool = True, plugins:
Optional[Set[Type[antispam.base_plugin.BasePlugin]]] = None)`

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
- **library** (*Library*) – The Library you are using.
- **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)

- **plugins** (*Set[Type[antislam.BasePlugin]]*) – A set for plugin lookups if you want to initialise plugins from an initial saved state. This should follow the format.

`{ClassReference}`

So for example:

```
1 class Plugin(BasePlugin):
2     pass
3
4 # Where you load ASH
5 await AntiSpamHandler.load_from_dict(..., ..., plugins={Plugin})
```

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)

This will simply ignore the saved state of plugins that don't have a `plugins` mapping.

async propagate(*message*) → Optional[Union[*antislam.dataclasses.core.CorePayload*, dict]]

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`

async 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

async 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.

async 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.

async `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.

This will return saved Plugin states where the Plugin has implemented the `save_to_dict` method.

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.

set_cache(*cache*: `antislam.abc.cache.Cache`) → `None`

Change the AntiSpamHandler internal cache to be the one provided.

```
1 bot.handler = AntiSpamHandler(bot)
2 cache = MongoCache(bot.handler, "Connection_url")
3 bot.handler.set_cache(cache)
```

Parameters `cache` (`Cache`) – The cache to change it to.

Raises `ValueError` – The provided cache was not of the expected type.

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

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

Wraps around `antislam.abc.Lib.visualizer()` as a convenience

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.mongo.MongoCache*
- *antispam.caches.redis.RedisCache*

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`.

Alternately, use the `AntiSpamHandler.set_cache` method.

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.

2.1 Redis Cache

Here is an example, note `RedisCache` needs a extra argument.

```
1 import discord
2 from discord.ext import commands
3 from redis import asyncio as aioredis
4
5 from antispam import AntiSpamHandler
6 from antispam.caches.redis import RedisCache
7
8 bot = commands.Bot(command_prefix="!", intents=discord.Intents.all())
9 bot.handler = AntiSpamHandler(bot)
10
11 redis = aioredis.from_url("redis://localhost")
12 redis_cache: RedisCache = RedisCache(bot.handler, redis)
13 bot.handler.set_cache(redis_cache)
```

2.2 MongoDB Cache

Here is an example, note MongoCache needs a extra argument.

```
1 import discord
2 from discord.ext import commands
3
4 from antispam import AntiSpamHandler
5 from antispam.caches.mongo import MongoCache
6
7 bot = commands.Bot(command_prefix="!", intents=discord.Intents.all())
8 bot.handler = AntiSpamHandler(bot)
9
10 my_cache = MongoCache(bot.handler, "Mongo connection url")
11 bot.handler.set_cache(my_cache)
```


POSSIBLE MODES

How to setup n use Antispam for different things.

Unless otherwise noted, ‘spam’ means messages that are basically the same.

3.1 Default

The default mode administers antispam on a per member per guild basis. This means that every single message sent by someone counts towards there spam threshold, this is great because it means people who spam in one channel and people who spam 1 message per channel both get caught.

3.2 Per Member Per Channel

The package also supports the ability to track spam per member per channel. This means that if you spam a single channel you will get punished, however if you send the same message in different channels your fine. You can set this using the following `Options(per_channel_spam=True)`

3.3 Max Message Amount

Unlike other methods, this punishes members if they simply send more then x messages within y time period. You can find this plugin here [*antispam.plugins.MaxMessageLimiter*](#)

EXAMPLE USAGES

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

4.1 Super duper basic bot

```
1 import discord
2 from discord.ext import commands
3
4 from antispam import AntiSpamHandler
5
6 bot = commands.Bot(command_prefix="!", intents=discord.Intents.all())
7 bot.handler = AntiSpamHandler(bot)
8
9
10 @bot.event
11 async def on_ready():
12     # On ready, print some details to standard out
13     print(f"-----\nlogged in as: {bot.user.name} : {bot.user.id}\n-----")
14
15
16 @bot.event
17 async def on_message(message):
18     await bot.handler.propagate(message)
19     await bot.process_commands(message)
20
21
22 if __name__ == "__main__":
23     bot.run("Bot Token Here")
```

4.2 Basic Hikari bot

```
1 import hikari
2 from antispam import AntiSpamHandler
3 from antispam.enums import Library
4
5 bot = hikari.GatewayBot(
6     token="..."
```

(continues on next page)

(continued from previous page)

```

7 )
8 handler = AntiSpamHandler(bot, library=Library.HIKARI)
9
10 @bot.listen()
11 async def ping(event: hikari.GuildMessageCreateEvent) -> None:
12     if event.is_bot or not event.content:
13         return
14
15     await handler.propagate(event.message)
16
17 bot.run()

```

4.3 How to use templating in a string

```

1 from discord.ext import commands
2
3 from antispam import AntiSpamHandler, Options
4
5 bot = commands.Bot(command_prefix="!")
6 bot.handler = AntiSpamHandler(bot, options=Options(ban_message="$MENTIONUSER you are_
↳ hereby banned from $GUILDNAME for spam!"))
7
8 @bot.event
9 async def on_ready():
10     print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")
11
12 @bot.event
13 async def on_message(message):
14     await bot.handler.propagate(message)
15     await bot.process_commands(message)
16
17 if __name__ == "__main__":
18     bot.run("Bot Token")

```

4.4 Cog Based Usage

```

1 from discord.ext import commands
2 from antispam import AntiSpamHandler
3
4 class AntiSpamCog(commands.Cog):
5     def __init__(self, bot):
6         self.bot = bot
7         self.bot.handler = AntiSpamHandler(self.bot)
8
9     @commands.Cog.listener()
10    async def on_ready(self):
11        print("AntiSpamCog is ready!\n-----\n")

```

(continues on next page)

(continued from previous page)

```

12     @commands.Cog.listener()
13     async def on_message(self, message):
14         await self.bot.handler.propagate(message)
15
16
17 def setup(bot):
18     bot.add_cog(AntiSpamCog(bot))

```

4.5 How to use templating in embeds

```

1 from discord.ext import commands
2
3 from antispam import AntiSpamHandler, Options
4
5 bot = commands.Bot(command_prefix="!")
6
7 warn_embed_dict = {
8     "title": "***Dear $USERNAME**",
9     "description": "You are being warned for spam, please stop!",
10    "timestamp": True,
11    "color": 0xFF0000,
12    "footer": {"text": "$BOTNAME", "icon_url": "$BOTAVATAR"},
13    "author": {"name": "$GUILDNAME", "icon_url": "$GUILDICON"},
14    "fields": [
15        {"name": "Current warns:", "value": "$WARNCOUNT", "inline": False},
16        {"name": "Current kicks:", "value": "$KICKCOUNT", "inline": False},
17    ],
18 }
19 bot.handler = AntiSpamHandler(bot, options=Options(guild_warn_message=warn_embed_dict))
20
21 @bot.event
22 async def on_ready():
23     print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")
24
25 @bot.event
26 async def on_message(message):
27     await bot.handler.propagate(message)
28     await bot.process_commands(message)
29
30 if __name__ == "__main__":
31     bot.run("Bot Token")

```

4.6 Custom Punishments

```

1  from discord.ext import commands
2
3  from antispam import AntiSpamHandler, Options
4  from antispam.plugins import AntiSpamTracker
5
6  bot = commands.Bot(command_prefix="!")
7  bot.handler = AntiSpamHandler(bot, options=Options(no_punish=True, message_duplicate_
8  ↪count=3))
9  bot.tracker = AntiSpamTracker(bot.handler, 5) # 5 Being how many 'punishment requests'
10 ↪before is_spamming returns True
11 bot.handler.register_plugin(bot.tracker)
12
13 @bot.event
14 async def on_ready():
15     # On ready, print some details to standard out
16     print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")
17
18 @bot.event
19 async def on_message(message):
20     await bot.handler.propagate(message)
21
22     if await bot.tracker.is_spamming(message):
23         # Insert code to mute the user
24
25         # Insert code to tell admins
26
27         # ETC
28         bot.tracker.remove_punishments(message)
29
30     await bot.process_commands(message)
31
32 if __name__ == "__main__":
33     bot.run("Bot Token")

```

PACKAGE LOGGING

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

5.1 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.

```
1 logging.basicConfig(
2     format="%(levelname)-7s | %(asctime)s | %(filename)12s:%(funcName)-12s | %(message)s
  ↳ ",
3     datefmt="%I:%M:%S %p %d/%m/%Y",
4     level=logging.INFO,
5 )
```

A more full example,

```
1 import logging
2
3 import discord
4 from discord.ext import commands
5
6 from antispam import AntiSpamHandler
7 from jsonLoader import read_json
8
9 logging.basicConfig(
10     format="%(levelname)s | %(asctime)s | %(module)s | %(message)s",
11     datefmt="%d/%m/%Y %I:%M:%S %p",
12     level=logging.INFO,
13 )
14
15 bot = commands.Bot(command_prefix="!", intents=discord.Intents.all())
16
17 file = read_json("token")
18
19 # Generally you only need/want AntiSpamHandler(bot)
20 bot.handler = AntiSpamHandler(bot, ignore_bots=False)
21
22
```

(continues on next page)

(continued from previous page)

```
23 @bot.event
24 async def on_ready():
25     # On ready, print some details to standard out
26     print(f"-----\nlogged in as: {bot.user.name} : {bot.user.id}\n-----")
27
28
29 @bot.event
30 async def on_message(message):
31     await bot.handler.propagate(message)
32     await bot.process_commands(message)
33
34
35 if __name__ == "__main__":
36     bot.run(file["token"])
```


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.

6.1 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`
- **\$MENTIONBOT** - This will attempt to mention your bot, uses `discord.Guild.me.mention`
- **\$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`

6.2 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**

6.3 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.

MIGRATING TO 1.0

The biggest change from 0.x.x to 1.x.x is that everything 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.

7.1 Changes

- **Extensions are now called plugins**
 - `from antispam.ext import ... -> from antispam.plugins import ...`
- `antispam.AntiSpamHandler` now takes an `antispam.dataclasses.options.Options` class rather than 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 `antispam.enums.IgnoreType` and `antispam.enums.ResetType`
- `antispam.AntiSpamHandler.propagate()` now returns `antispam.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

7.2 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
 - `antispam.AntiSpamHandler.add_guild_log_channel()`
 - `antispam.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 `antispam.abc.Cache`
- Now features an easy way to clean up your cache. See `antispam.AntiSpamHandler.clean_cache()`
- **New plugins:**
 - `antispam.plugins.AntiMassMention` - To stop people spam pinging

- `antislam.plugins.Stats` - For general package stats
- `antislam.plugins.AdminLogs` - An easy way to get evidence on punishments
- Plugins now have direct access to storage within the cache. You should be interacting with `antislam.PluginCache` for this.
- Plugins now support blacklisting to stop runs on certain guilds. See Plugin Blacklisting under Package Plugin System
- Roughly 150% faster than 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

7.3 Fixes

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

CHANGELOG

A changelog that should be fairly up to date feature wise.

8.1 1.3.0 -> 1.4.0

Support for Redis as a built in cache backend.

8.2 1.2.11 -> 1.3.0

8.2.1 New:

- The AdminLogs plugin now supports custom punishments via a new argument when initializing it. See the plugin for further info.

8.2.2 Fixes:

- Upgraded the nextcord Lib to support the latest release. This is not backwards compatible.
- AntiSpamTracker should now be library agnostic.
- AntiSpamTrackerSubclass example is now up to date.
- A regression in clean cache where it checked Options rather than the current anti spam handlers options

8.2.3 Changes:

- All usages of user in AntiSpamTracker are now member
- **MaxMessageLimiter:**
 - New hard cap of 50
 - Is now guild wide
 - Punishment is a timeout for 5 minutes
- Default punishment scheme is to timeout members.

8.2.4 Removed:

- DPY as a default for the library options in init and load_from_dict.
- The ability to use names for ignored channels and roles.

8.3 1.2.8 -> 1.2.9

Fixed timeouts 'stacking' when they shouldn't have.

Resolved some issues with discord.py and forks where default avatars would result in errors.

8.4 1.2.(3-7) -> 1.2.8

Fixed MongoCache and MyCustomTracker. These are also both now fully unit-tested.

8.5 1.2.2 -> 1.2.3

Fixed the AdminLogs plugin for timeouts.

8.6 1.2.1 -> 1.2.2

The big feature of this release is the new punishment handling using discord timeouts. You can opt into these by passing `use_timeouts=True` as an option.

This will become the default scheme in version 1.3.0.

8.6.1 New:

- 1) Added the `times_timed_out` field to `Member`'s
- 2) Added `member_was_timed_out` field to `CorePayload`
- 3) **Added the following fields to `Options`**
 - `use_timeouts`
 - `member_timeout_message`
 - `guild_log_timeout_message`
 - `member_failed_timeout_message`
 - `member_timeout_message_delete_after`
 - `guild_log_timeout_message_delete_after`

8.6.2 Fixes:

- 1) **Option** attributes missing documentation
- 2) **How the core handler used Options** Previously per-guild options were ignored in most situations, this has been fixed.

8.6.3 Deprecated:

- 1) **Library defaults** In version 1.3.0 you will need to explicitly define the library you are running AntiSpam with.
- 2) **MaxMessages Plugin** It will be getting reworked in version 1.3.0, if you wish to preserve current behaviour save it locally.
- 3) Support for `.name` lookups in ignored channels & roles

8.6.4 Changes:

- 1) **Lib impl changes** Reworked the lib folder to provide explicit fork support and reduce duplicated code.
- 2) **Options.is_per_channel_per_guild defaults** This was changed to `False` to preserve the current behaviour once it is actually implemented.

8.7 1.2.0 -> 1.2.1

8.7.1 Fixes:

Mongo is no longer required when not using the `MongoCache`

8.8 1.1.3 -> 1.2.0

8.8.1 New:

1. **Added support for `Pincer`.** You can use this by passing the `library=Library.Pincer` enum to your `AntiSpamHandler` during initialization.
2. **New message templating option.** `$MENTIONBOT` to mention your bot.
3. **A method for changing caches.** `AntiSpamHandler.set_cache(new_cache)`
4. **A new cache.** `MongoCache(AntiSpamHandler, "Mongo connection url")`

8.8.2 Fixed:

- Hikari regressions
- Some misc bugs

8.8.3 Changes:

- **Modified Lib interface** Check it out, its more DRY now.
- **Modified Member** `_in_guild` is now `internal_is_in_guild`

8.8.4 Removed:

- **Extra installs for:** DPY and hikari

8.9 1.1.2 -> 1.1.3

Backwards compatible changes:

- **Closed issue #73 on Github, this means you can now save plugin states.**
 - Note only the shipped Stats plugin currently saves it's state.

ENUM REFERENCE

```
class antispam.enums.IgnoreType(value)
```

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(value)
```

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

```
class antispam.enums.Library(value)
```

An enum to denote which type of API wrapper you are intending on using this with. Defaults to DPY.

Notes

Default behaviour will be removed in 1.3.0

CUSTOM will not set a library handler. You will need to set one yourself.

CUSTOM = 8

DISNAKE = 5

DPY = 1

ENHANCED_DPY = 4

HIKARI = 2

NEXTCORD = 6

PINCER = 3

PYCORD = 7

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 how to include variables, please see the [Message Templating](#) docs.

```

class antispy.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_log_warn_message: Union[str, dict] =
'$MEMBERNAME was warned for
spamming/sending duplicate messages.',
guild_log_kick_message: Union[str, dict] =
'$MEMBERNAME was kicked for
spamming/sending duplicate messages.',
guild_log_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.",
use_timeouts: bool = True,
member_timeout_message: Union[str, dict] =
'Hey $MENTIONMEMBER, you are being
timed out from $GUILDNAME for
spamming/sending duplicate messages.',
member_failed_timeout_message: Union[str,
dict] = "I failed to punish you because I lack
permissions, but still you shouldn't spam.",
guild_log_timeout_message: Union[str, dict] =
'$MEMBERNAME was timed out for
spamming/sending duplicate messages.',
guild_log_timeout_message_delete_after:
Optional[int] = None,
member_timeout_message_delete_after:
Optional[int] = None,
guild_log_ban_message_delete_after:
Optional[int] = None,
guild_log_kick_message_delete_after:
Optional[int] = None,
member_ban_message_delete_after:
Optional[int] = None,
guild_log_warn_message_delete_after:
Optional[int] = None,
member_kick_message_delete_after:
Optional[int] = None,
member_warn_message_delete_after:
Optional[int] = None,
ignored_members=NOTHING,
ignored_channels=NOTHING,
ignored_roles=NOTHING,
ignored_guilds=NOTHING, delete_spam: bool

```

Parameters

- **use_timeouts** (*bool*) – Default: None

If True, use timeouts as the punishment scheme. If False, then the default punishment scheme of warn, kick, ban is used.

This will be set to default to True in version 1.3.0

- **warn_threshold** (*int*) – Default: 3

This is the amount of duplicates within `message_interval` that will result in a warning.

- **kick_threshold** (*int*) – Default: 2

This is the amount of warns required before a kick is the next punishment. I.e. After 2 warns, you will get kicked.

- **ban_threshold** (*int*) – Default: 2

This is the amount of kicks required before a ban is the next punishment. I.e. After 2 kicks, you will get banned.

- **message_interval** (*int*) – Default: 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.

- **message_duplicate_count** (*int*) – Default: 5

The amount of duplicate messages needed within `message_interval` to trigger a punishment. I.e. Once you've sent 5 'spam' messages you'll get punished.

- **message_duplicate_accuracy** (*int*) – Default: 90

How 'close' messages need to be to be registered as duplicates (Out of 100) You can test this with the code:

```
from fuzzywuzzy import fuzz
fuzz.token_sort_ratio("message one", "message two")
```

- **guild_log_warn_message** (*Union[str, dict]*) – Default: \$MEMBERNAME was warned for spamming/sending duplicate messages.

The message to be sent in the guild when someone is warned. Please see the note at the bottom.

- **guild_log_kick_message** (*Union[str, dict]*) – Default: \$MEMBERNAME was kicked for spamming/sending duplicate messages.

The message to be sent in the guild when someone is kicked. Please see the note at the bottom.

- **guild_log_ban_message** (*Union[str, dict]*) – Default: \$MEMBERNAME was banned for spamming/sending duplicate messages.

The message to be sent in the guild when someone is banned. Please see the note at the bottom.

- **guild_log_timeout_message** (*Union[str, dict]*) – Default: \$MEMBERNAME was timed out for spamming/sending duplicate messages.

The message to be sent in the guild when someone is timed out. Please see the note at the bottom.

- **member_warn_message** (*Union[str, dict]*) – Default: Hey \$MENTIONMEMBER, please stop spamming/sending duplicate messages.

The message to be sent in the guild when a member is warned.

- **member_kick_message** (*Union[str, dict]*) – Default: Hey \$MENTIONMEMBER, you are being kicked from \$GUILDNAME for spamming/sending duplicate messages.

The message to be sent to the member who is being kicked.

- **member_ban_message** (*Union[str, dict]*) – Default: Hey \$MENTIONMEMBER, you are being banned from \$GUILDNAME for spamming/sending duplicate messages.

The message to be sent to the member who is being banned.

- **member_timeout_message** (*Union[str, dict]*) – Default: Hey \$MENTIONMEMBER, you are being timed out from \$GUILDNAME for spamming/sending duplicate messages.

The message to be sent to the member who is being timed out.

- **member_failed_kick_message** (*Union[str, dict]*) – Default: I failed to punish you because I lack permissions, but still you shouldn't spam.

The message to be sent if kicking the member fails.

- **member_failed_ban_message** (*Union[str, dict]*) – Default: I failed to punish you because I lack permissions, but still you shouldn't spam.

The message to be sent if banning the member fails.

- **member_failed_timeout_message** (*Union[str, dict]*) – Default: I failed to punish you because I lack permissions, but still you shouldn't spam.

The message to be sent if kicking the member fails.

- **guild_log_warn_message_delete_after** (*int*) – Default: None

How many seconds after sending the guild warn message to delete it.

- **guild_log_kick_message_delete_after** (*int*) – Default: None

How many seconds after sending the guild kick message to delete it.

- **guild_log_ban_message_delete_after** (*int*) – Default: None

How many seconds after sending the guild ban message to delete it.

- **guild_log_timeout_message_delete_after** (*int*) – Default: None

How many seconds after sending the guild timeout message to delete it.

- **member_warn_message_delete_after** (*int*) – Default: None

How many seconds after sending the member warn message to delete it.

- **member_kick_message_delete_after** (*int*) – Default: None

How many seconds after sending the member kick message to delete it.

- **member_ban_message_delete_after** (*int*) – Default: None
How many seconds after sending the member ban message to delete it.
- **member_timeout_message_delete_after** (*int*) – Default: None
How many seconds after sending the member timeout message to delete it.
- **ignored_members** (*Set[int]*) – Default: Empty Set
A Set of members to ignore messages from. Set this with *antislam.AntiSpamHandler.add_ignored_item()* Remove members with *antislam.AntiSpamHandler.remove_ignored_item()*
- **ignored_channels** (*Set[int]*) – Default: Empty Set
A Set of channels to ignore messages in. Set this with *antislam.AntiSpamHandler.add_ignored_item()* Remove channels with *antislam.AntiSpamHandler.remove_ignored_item()*
- **ignored_roles** (*Set[int]*) – Default: Empty Set
A Set of roles to ignore messages from. Set this with *antislam.AntiSpamHandler.add_ignored_item()* Remove roles with *antislam.AntiSpamHandler.remove_ignored_item()*
- **ignored_guilds** (*Set[int]*) – Default: Empty Set
A Set of guilds to ignore messages in. Set this with *antislam.AntiSpamHandler.add_ignored_item()* Remove guilds with *antislam.AntiSpamHandler.remove_ignored_item()*
- **delete_spam** (*bool*) – Default: 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 delete all messages marked as spam, and this means an api call per message.
- **ignore_bots** (*bool*) – Default: True
Should bots bypass anti-spam?
- **warn_only** (*bool*) – Default: False
Whether or not to only warn users, this means it will not kick or ban them.
- **no_punish** (*bool*) – Default: 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.
Use *antislam.plugins.AntiSpamTracker* with this mode for best affect.
- **mention_on_embed** (*bool*) – Default: True
If the message your trying to send is an embed, also send some content to mention the person being punished.

- **delete_zero_width_chars** (*bool*) – Default: test

Should zero width characters be removed from messages. Useful as otherwise it helps people bypass antispam measures.

- **per_channel_spam** (*bool*) – Default: False

Track spam as per channel, rather than per guild. I.e. False implies spam is tracked as Per Member Per Guild True implies Per Member Per Channel

- **addons** (*Dict*) – Default: Empty Dict

Use-able storage for plugins to store Options

Notes

Guild log messages will **only** send if `antispam.dataclasses.guild.Guild.log_channel_id` is set. You can set it with `antispam.AntiSpamHandler.add_guild_log_channel()`

```
__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_log_warn_message: Union[str, dict] =
         '$MEMBERNAME was warned for spamming/sending duplicate messages.',
         guild_log_kick_message: Union[str, dict] = '$MEMBERNAME was kicked for
         spamming/sending duplicate messages.', guild_log_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.", use_timeouts: bool = True,
         member_timeout_message: Union[str, dict] = 'Hey $MENTIONMEMBER, you are
         being timed out from $GUILDNAME for spamming/sending duplicate messages.',
         member_failed_timeout_message: Union[str, dict] = "I failed to punish you because I
         lack permissions, but still you shouldn't spam.", guild_log_timeout_message:
         Union[str, dict] = '$MEMBERNAME was timed out for spamming/sending duplicate
         messages.', guild_log_timeout_message_delete_after: Optional[int] = None,
         member_timeout_message_delete_after: Optional[int] = None,
         guild_log_ban_message_delete_after: Optional[int] = None,
         guild_log_kick_message_delete_after: Optional[int] = None,
         member_ban_message_delete_after: Optional[int] = None,
         guild_log_warn_message_delete_after: Optional[int] = None,
         member_kick_message_delete_after: Optional[int] = None,
         member_warn_message_delete_after: Optional[int] = None,
         ignored_members=NOTHING, ignored_channels=NOTHING,
         ignored_roles=NOTHING, ignored_guilds=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 = False, addons:
         Dict[str, Any] = NOTHING) → None
```

Method generated by attrs for class Options.

addons

ban_threshold
delete_spam
delete_zero_width_chars
guild_log_ban_message
guild_log_ban_message_delete_after
guild_log_kick_message
guild_log_kick_message_delete_after
guild_log_timeout_message
guild_log_timeout_message_delete_after
guild_log_warn_message
guild_log_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_failed_timeout_message
member_kick_message
member_kick_message_delete_after
member_timeout_message
member_timeout_message_delete_after
member_warn_message
member_warn_message_delete_after
mention_on_embed
message_duplicate_accuracy
message_duplicate_count
message_interval
no_punish
per_channel_spam
use_timeouts

`warn_only`

`warn_threshold`

COREPAYLOAD REFERENCE

You should not be creating this object yourself.

```
class antisperm.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_was_timed_out: 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 `antisperm.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_was_timed_out** (`bool`) – If the default punishment handler timed out this member
- **member_should_be_punished_this_message** (`bool`) – If AntiSpamHandler thinks this member should receive some form of punishment this message. Useful for `antisperm.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_was_timed_out: 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.

```
after_invoke_extensions: Dict[str, Any]
```

```
member_duplicate_count: int
```

```
member_kick_count: int
```

```
member_should_be_punished_this_message: bool
member_status: str
member_warn_count: int
member_was_banned: bool
member_was_kicked: bool
member_was_timed_out: bool
member_was_warned: bool
pre_invoke_extensions: Dict[str, Any]
```

PACKAGE PLUGIN SYSTEM

This package features 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.

12.1 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.

12.2 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 order 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

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`

13.1 Pre-invoke Schema

```
1 from antispan import BasePlugin
2
3 class Placeholder(BasePlugin):
4     def __init__(self):
5         self.is_pre_invoke = True
6
7     async def propagate(self, message: discord.Message) -> dict:
8         # 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.

13.2 After-invoke Schema

```
1 from antispan import BasePlugin
2
3 class Placeholder(BasePlugin):
4     def __init__(self):
5         self.is_pre_invoke = False
6
7     async def propagate(self, message: discord.Message, propagate_data: CorePayload) -> dict:
8         # 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`

13.3 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

PLUGIN-CACHE INTERACTION

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

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

Parameters

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

async `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

async `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

async `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

async 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

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 doesn't) 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

member_tracking

The underlying cache mechanism for data storage

Type *PluginCache*

```
__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 member 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

async do_punishment(*message*, **args*, ***kwargs*) → **None**

This only exists for if the member wishes to subclass this class and implement there own logic for punishments here.

Parameters *message* – The message to extract the guild and member from

Notes

This does nothing unless you subclass and implement it yourself.

async get_member_count(*message*) → **int**

Returns how many messages that are still ‘valid’ (counted as spam) a certain member has

Parameters *message* – The message from which to extract member

Returns How many times this member 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

async is_spamming(*message*) → **bool**

Given a message, deduce and return if a member is classed as ‘spamming’ or not based on *punish_min_amount*

Parameters *message* – The message to extract guild and member from

Returns True if the User is spamming else False

Return type **bool**

member_tracking

async propagate(*message*, *data*: *Optional[antispy.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

async remove_outdated_timestamps(*data*: *List*, *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 member

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

async remove_punishments(*message*)

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

Parameters `message` – The message to extract member from

Raises `TypeError` – Invalid arg

Notes

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

async `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`

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.

channel_id: *int*

guild_id: *int*

is_overall_punishment: *bool*

member_id: *int*

```
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.

```
1 data = await AntiSpamHandler.propagate(message)
2 return_item: Union[dict, MassMentionPunishment] = data.after_invoke_plugins[
  ↳ "AntiMassMention"]
3
4 if isinstance(return_item, MassMentionPunishment):
5     # 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*

async 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.

STATISTICS PLUGIN

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

```
1 from discord.ext import commands
2
3 from antispam import AntiSpamHandler
4 from antispam.ext import Stats
5
6 bot = commands.Bot(command_prefix="!")
7 bot.handler = AntiSpamHandler(bot, no_punish=True)
8 bot.stats = Stats(bot.handler)
9 bot.handler.register_extension(bot.stats)
10
11 # We don't want to collect stats on guild 12345
12 # So lets ignore it on this plugin
13 bot.stats.blacklisted_guilds.add(12345)
14
15
16 @bot.event
17 async def on_ready():
18     # On ready, print some details to standard out
19     print(f"-----\nLogged in as: {bot.user.name} : {bot.user.id}\n-----")
20
21
22 if __name__ == "__main__":
23     bot.run("Bot Token")
```

```
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)
```

```
injectable_nonce = 'Issa me, Mario!'
```

```
async classmethod load_from_dict(anti_spam_handler:
                                antispam.anti_spam_handler.AntiSpamHandler, data: Dict)
```

```
async 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`

async save_to_dict() → Dict

Saves the plugins state to a Dict

Returns The current plugin state as a dictionary.

Return type Dict

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, punishment_type: Optional[Union[str, Callable]] = None)
```

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, punishment_type:
          Optional[Union[str, Callable]] = None)
```

Parameters

- **handler** ([AntiSpamHandler](#)) – Our AntiSpamHandler instance
- **log_location** ([str](#)) – The directory to store logs in, relative from the caller location. This directory should be empty or only contain previous output from this plugin.
- **punishment_type** ([Optional\[Union\[str, Callable\]\]](#)) – This will be used if sending logs for custom punishments.

You can also provide a [Callable](#) which should return a string to be used as the punishment type. This function will be called with 2 arguments. Argument 1 is the message, argument 2 is [CorePayload](#)

Notes

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

```
async propagate(message, data: Optional[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](#)

MAXMESSAGELIMITER PLUGIN

This plugin implements a hard cap for the amount of messages a member can send within the handlers timeframe before losing send message perms.

```
class antispam.plugins.MaxMessageLimiter(handler: antispam.anti_spam_handler.AntiSpamHandler,  
                                         hard_cap: int = 50, message_interval: Optional[int] = None)
```

This plugin implements a hard cap for the amount of messages a member can send within the handlers timeframe before losing send message perms.

Notes

This only works with the initial message_interval, no updates or guild specifics. If this is something you want let me know on discord and I might make it.

This is also guild wide.

```
__init__(handler: antispam.anti_spam_handler.AntiSpamHandler, hard_cap: int = 50, message_interval:  
         Optional[int] = None)
```

Parameters

- **handler** ([AntiSpamHandler](#)) – The handler to extract cache from
- **hard_cap** (*int*, *Optional*) – The hard cap for the amount of messages you can send within message_interval

Defaults to 50

- **message_interval** (*int*, *Optional*) – The period of time in milliseconds which messages should be treated as valid.

Defaults to your AntiSpamHandler options message_interval

```
async do_punishment(member: antispam.dataclasses.member.Member, message) → None
```

This method gets called and handles punishments, override it to change punishments.

Parameters

- **member** ([Member](#)) – The member to punish
- **message** – A message to get info from

```
async propagate(message, data: Optional[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`

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. This 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.

20.1 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.

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 guaranteed 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

async 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

async delete_guild(guild_id: int) → None

Removes a guild from the cache. This should also remove all members.

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

Notes

This fails silently.

async 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.

async drop() → `None`

Drops the entire cache, deleting everything contained within.

async get_all_guilds() → `AsyncIterable[antispam.dataclasses.guild.Guild]`

Returns a generator containing all cached guilds

Yields *Guild* – A generator of all stored guilds

async 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

async 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

async 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

async 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.

async 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

Notes

This shouldn't raise an error if the member doesn't exist.

async 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

async 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.

You also should to subclass [antispam.libs.shared.base.Base](#) as it implements a lot of shared functionality.

async 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.

Notes

Should error on the following:

- If not an instance of the library's message class
- If in dm's
- If the message is from yourself (the bot)
- If `self.handler.options.ignore_bots` is True and the message is from a bot
- If the guild id is in `self.handler.options.ignored_guilds`
- If the member id is in `self.handler.options.ignored_members`
- If the channel id is in `self.handler.options.ignored_channels`
- If any of the member's roles (id) are in `self.handler.options.ignored_roles`

`PropagateData.has_perms_to_make_guild` should be True if the member has permissions to kick and ban members

Parameters **message** ([Union\[discord.Message, hikari.messages.Message, pincer.objects.Embed\]](#)) – 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 gets returned from within propagate itself.

async create_message(message) → *antisпам.dataclasses.message.Message*

Given a message to extract data from, create and return a Message class.

The following should be dumped together as content. Order doesn't matter as long as it's consistent.

- All sticker urls for stickers in the message
- The actual message content
- All embeds cast to a string via `embed_to_string`

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

Returns The flushed out message

Return type *Message*

Raises *InvalidMessage* – If it couldn't create a message, I.e message only contained attachments

Warning: This is REQUIRED to be inline with the end options.

```
1 if self.handler.options.delete_zero_width_chars:
2     content = (
3         content.replace("u200B", "")
4         .replace("u200C", "")
5         .replace("u200D", "")
6         .replace("u200E", "")
7         .replace("u200F", "")
8         .replace("uFEFF", "")
9     )
```

async delete_member_messages(member: *antisпам.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

If you need to fetch the channel, cache it please.

async delete_message(message) → *None*

Given a message, call and handle the relevant deletion contexts.

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

Notes

This should handle given errors silently.

async 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 The embed

Return type *Union[discord.Embed, hikari.embeds.Embed, pincer.objects.Embed]*

Notes

This is implemented in *antispam.libs.shared.base.Base* so assuming you subclass it you don't need to create this.

async dict_to_lib_embed(data: *Dict*)

Create an Embed object using a dictionary and return that.

Parameters **data** (*Dict*) – The data to create the embed from

Returns The dictionary as an embed

Return type *Union[discord.Embed, hikari.embeds.Embed, pincer.objects.Embed]*

async embed_to_string(embed) → *str*

Given an embed, return a string representation.

This string representation should include the following. Ordering does not matter as long as it is consistent.

- The embeds title
- The embeds description
- The embeds footer text
- The embeds author name
- All names + values for embed fields

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

Returns The embed as a string

Return type *str*

Notes

This is implemented in `antispam.libs.shared.base.Base` so assuming you subclass it you don't need to create this.

async `get_channel_by_id(channel_id: int)`

Returns the given channel object for the id.

async `get_channel_from_message(message)`

Given a message, return the channel object it was sent in.

async `get_channel_id(message) → int`

Returns the id of the channel this message was sent in.

get_file(path: str)

Returns a file object for the given path.

For example, in discord.py this is `discord.File`

async `get_guild_id(message) → int`

Returns the id of the guild this message was sent in.

async `get_member_from_message(message)`

Given the message, return the Author's object

Parameters `message` (`Union[discord.Message, hikari.messages.Message, pincer.objects.Embed]`) – The message to extract it from

Returns The member object

Return type `Union[discord.Member, hikari.guilds.Member, ...]`

async `get_message_mentions(message) → List[int]`

Returns all the mention id's from a message.

This should include:

- People
- Roles
- Channels

async `get_substitute_args(message) → antispam.libs.shared.substitute_args.SubstituteArgs`

Given a message, return the relevant class filled with data for usage within `substitute_args`

Parameters `message` (`Union[discord.Message, hikari.messages.Message, pincer.objects.UserMessage]`) – The message we are using for creation

Returns A dataclass with the relevant data for substitution.

Return type `SubstituteArgs`

is_dm(message) → bool

Returns True if this message occurred in a dm, False otherwise.

Parameters `message` – The discord message this is called on.

async `is_member_currently_timed_out(member) → bool`

Given the libraries member object, return True if they are currently timed out or False otherwise.

Parameters `member` (`Union[discord.member, hikari.guilds.Member]`) – The member to check against.

Returns True if timed out, otherwise False

Return type `bool`

async lib_embed_as_dict(*embed*) → Dict

Given the relevant Embed object for your library, return it in dictionary form.

Parameters *embed* (Union[discord.Embed, hikari.embeds.Embed, pincer.objects.Embed]) – The embed

Returns The embed as a dictionary

Return type `dict`

async 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*: Optional[int] = None, *channel_delete_after*: Optional[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, pincer.objects.UserMessage]) – Where we get everything from :)
- **user_message** (Union[str, discord.Embed, hikari.embeds.Embed, pincer.objects.Embed]) – A message to send to the user who is being punished
- **guild_message** (Union[str, discord.Embed, hikari.embeds.Embed, pincer.objects.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

Returns Did the punishment succeed?

Return type `bool`

Notes

Due to early design decisions, this will only ever support kicking or banning. A pain for you and I.

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

Sends a message to the guilds log channel

Parameters

- **guild** (Guild) – The guild we wish to send this too
- **message** (Union[str, discord.Embed, hikari.embeds.Embed, pincer.objects.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.channels.GuildTextChannel, pincer.objects.Channel]`) – 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.

If no log channel, don't send anything.

async send_message_to_(*target*, *message*, *mention*: *str*, *delete_after_time*: *Optional[int]* = None) → None
Send the given message to the target.

Parameters

- **target** (`Union[discord.abc.Messageable, hikari.channels.TextableChannel]`) – Where to send the message.
Types are unknown for Pincer at this time.
- **message** (`Union[str, discord.Embed, hikari.embeds.Embed, pincer.objects.Embed]`) – The message to send
- **mention** (*str*) – A string denoting a raw mention of the punished user
- **delete_after_time** (*Optional[int]*) – When to delete the message after

Notes

This should implement Options.mention_on_embed

async 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, pincer.objects.UserMessage]`) – 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*

Notes

This is implemented in `antislam.libs.shared.base.Base` so assuming you subclass it you don't need to create this.

async timeout_member(*member*, *original_message*, *until*: `datetime.timedelta`) → `None`

Timeout the given member.

Parameters

- **member** (`Union[discord.Member, hikari.guilds.Member]`) – The member to time-out
- **original_message** – The message being propagated
- **until** (`datetime.timedelta`) – How long to time them out for.

Raises

- **UnsupportedAction** – Timing out members is not supported.
- **MissingGuildPermissions** – Can't time this member out

async 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, pincer.objects.UserMessage]`) – 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 A template complete message ready for sending

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

Notes

This is implemented in `antislam.libs.shared.base.Base` so assuming you subclass it you don't need to create this.

async 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]`

Notes

This is implemented in `antispam.libs.shared.base.Base` so assuming you subclass it you don't need to create this.

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 `BaseASException` which provides functionality for error messages

The MIT License (MIT)

Copyright (c) 2020-Current Skelmis

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

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

A base exception handler for the ASH ecosystem.

`__init__(*args)`

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.ExistingEntry(*args)`

An entry was already found in the timed cache with this key.

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.InvalidMessage(*args)`

Could not create a use-able message for the given message.

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 and include traceback.

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

This class has no 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.**NonExistentEntry**(*args)

No entry found in the timed cache with this key.

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*)

exception antispam.exceptions.**UnsupportedAction**(*args)

The attempt action is unsupported.

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: Optional[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: Optional[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

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 a way of storing Messages as well as maintaining the required data to track and punish spammers.

Please note, if you plan on working with any of the duplicate counter values you **need** to minus 1 in order to get the **actual** value. This is due to the fact the counter starts at 1 since we don't mark the first message as spam due to some internal conflicts.

```
class antispam.dataclasses.member.Member(id: int, guild_id: int, warn_count: int = 0, kick_count: int = 0,
                                          times_timed_out: int = 0, duplicate_counter: int = 1,
                                          duplicate_channel_counter_dict: Dict[int, int] = NOTHING,
                                          internal_is_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, times_timed_out: int = 0,
          duplicate_counter: int = 1, duplicate_channel_counter_dict: Dict[int, int] = NOTHING,
          internal_is_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

internal_is_in_guild

kick_count

messages

times_timed_out

warn_count

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

REDISCACHE REFERENCE

A caching option within the standard package.

Furthermore, refer to [*antispam.abc.Cache*](#) for protocol implementation.

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*)

async 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

async delete_guild(*guild_id*: int) → None

Removes a guild from the cache. This should also remove all members.

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

Notes

This fails silently.

async 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.

async 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

async 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

async 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

async 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.

async 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

Notes

This shouldn't raise an error if the member doesn't exist.

async 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

async 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

MONGOCACHE REFERENCE

Furthermore, refer to [*antispam.abc.Cache*](#) for protocol implementation.

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

LIB BASE OBJECT

The generic feature class which all Lib's should subclass

class `antispam.libs.shared.base.Base`

A base Library feature class which implements shared functionality.

async `dict_to_embed(data: dict, message, warn_count: int, kick_count: int)`

async `dict_to_lib_embed(data: Dict)`

Parameters `data` (*dict*) – The embed as a dictionary, used to build a an embed object for your library.

Returns Your libraries embed object.

Return type Any

Warning: Lib classes must implement this.

async `embed_to_string(embed) → str`

async `get_substitute_args(message) → antispam.libs.shared.substitute_args.SubstituteArgs`

Parameters `message` – Message used to create SubstituteArgs

Returns

Return type *SubstituteArgs*

Warning: Lib classes must implement this.

async `lib_embed_as_dict(embed) → Dict`

Parameters `embed` – Your libraries embed object.

Returns The embed in dict form

Return type *dict*

Warning: Lib classes must implement this.

```
async substitute_args(content, message, warn_count: int, kick_count: int) → str
async transform_message(content: Union[str, dict], message, warn_count: int, kick_count: int)
async visualizer(content: str, message, warn_count: int = 1, kick_count: int = 2)
```

SUBSTITUTEARGS OBJECT

An attrs dataclass used to pass the required information around easily in order for Base to work.

```
class antisпам.libs.shared.substitute_args.SubstituteArgs(member_id: int, member_name: str,
                                                         member_avatar: str, bot_id: int,
                                                         bot_name: str, bot_avatar: str, guild_id:
                                                         int, guild_name: str, guild_icon: str)

    __init__(member_id: int, member_name: str, member_avatar: str, bot_id: int, bot_name: str, bot_avatar:
            str, guild_id: int, guild_name: str, guild_icon: str) → None
        Method generated by attrs for class SubstituteArgs.

    bot_avatar
    bot_id
    bot_name
    guild_icon
    guild_id
    guild_name
    member_avatar
    member_id
    member_name
    property mention_bot: str
    property mention_member: str
    property timestamp_now: str
    property timestamp_today: str
```


PLUGIN BASEPLUGIN OBJECT

The base class for all plugins.

```
class antispam.BasePlugin(is_pre_invoke=True)
```

```
    __init__(is_pre_invoke=True) → None
```

```
    async classmethod load_from_dict(anti_spam_handler, data: Dict)
```

Loads this plugin from a saved state.

Parameters

- **anti_spam_handler** ([AntiSpamHandler](#)) – The AntiSpamHandler instance
- **data** (*Dict*) – The data to load the plugin from

```
    async propagate(message, data: Optional[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`

```
    async save_to_dict() → Dict
```

Saves the plugins state to a Dict

Returns The current plugin state as a dictionary.

Return type `Dict`

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.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

a

`antispam.exceptions`, [69](#)

Symbols

__init__() (antispam.AntiSpamHandler method), 3
 __init__() (antispam.BasePlugin method), 91
 __init__() (antispam.CorePayload method), 37
 __init__() (antispam.PluginCache method), 43
 __init__() (antispam.caches.MemoryCache method), 79
 __init__() (antispam.dataclasses.guild.Guild method), 71
 __init__() (antispam.dataclasses.member.Member method), 73
 __init__() (antispam.dataclasses.message.Message method), 75
 __init__() (antispam.dataclasses.options.Options method), 34
 __init__() (antispam.dataclasses.propagate_data.PropagateData method), 85
 __init__() (antispam.exceptions.BaseASException method), 69
 __init__() (antispam.exceptions.PropagateFailure method), 70
 __init__() (antispam.libs.shared.substitute_args.SubstituteArgs attribute), 89
 __init__() (antispam.plugins.AdminLogs method), 53
 __init__() (antispam.plugins.AntiMassMention method), 49
 __init__() (antispam.plugins.MaxMessageLimiter method), 55
 __init__() (antispam.plugins.Stats method), 51

A

add_guild_log_channel() (antispam.AntiSpamHandler method), 3
 add_guild_options() (antispam.AntiSpamHandler method), 4
 add_ignored_item() (antispam.AntiSpamHandler method), 4
 add_message() (antispam.abc.Cache method), 59
 add_message() (antispam.caches.MemoryCache method), 79
 addons (antispam.dataclasses.guild.Guild attribute), 71

addons (antispam.dataclasses.member.Member attribute), 73
 addons (antispam.dataclasses.options.Options attribute), 34
 AdminLogs (class in antispam.plugins), 53
 after_invoke_extensions (antispam.CorePayload attribute), 37
 AntiMassMention (class in antispam.plugins), 49
 antispam.exceptions module, 69
 AntiSpamHandler (class in antispam), 3
 author_id (antispam.dataclasses.message.Message attribute), 75

B

ban_data_threshold (antispam.dataclasses.options.Options attribute), 35
 Base (class in antispam.libs.shared.base), 87
 BaseASException, 69
 BasePlugin (class in antispam), 91
 bot_avatar (antispam.libs.shared.substitute_args.SubstituteArgs attribute), 89
 bot_id (antispam.libs.shared.substitute_args.SubstituteArgs attribute), 89
 bot_name (antispam.libs.shared.substitute_args.SubstituteArgs attribute), 89

C

Cache (class in antispam.abc), 59
 CHANNEL (antispam.enums.IgnoreType attribute), 27
 channel_id (antispam.dataclasses.message.Message attribute), 75
 channel_id (antispam.plugins.MassMentionPunishment attribute), 49
 check_message_can_be_propagated() (antispam.abc.Lib method), 61
 clean_cache() (antispam.AntiSpamHandler method), 4
 content (antispam.dataclasses.message.Message attribute), 75
 CorePayload (class in antispam), 37
 create_message() (antispam.abc.Lib method), 61

`creation_time` (*antispam.dataclasses.message.Message* attribute), 75
`CUSTOM` (*antispam.enums.Library* attribute), 27

D

`delete_guild`() (*antispam.abc.Cache* method), 59
`delete_guild`() (*antispam.caches.MemoryCache* method), 79
`delete_member`() (*antispam.abc.Cache* method), 59
`delete_member`() (*antispam.caches.MemoryCache* method), 79
`delete_member_messages`() (*antispam.abc.Lib* method), 62
`delete_message`() (*antispam.abc.Lib* method), 62
`delete_spam` (*antispam.dataclasses.options.Options* attribute), 35
`delete_zero_width_chars` (*antispam.dataclasses.options.Options* attribute), 35
`dict_to_embed`() (*antispam.abc.Lib* method), 63
`dict_to_embed`() (*antispam.libs.shared.base.Base* method), 87
`dict_to_lib_embed`() (*antispam.abc.Lib* method), 63
`dict_to_lib_embed`() (*antispam.libs.shared.base.Base* method), 87
`DISNAKE` (*antispam.enums.Library* attribute), 27
`do_punishment`() (*antispam.plugins.MaxMessageLimiter* method), 55
`DPY` (*antispam.enums.Library* attribute), 27
`drop`() (*antispam.abc.Cache* method), 60
`drop`() (*antispam.caches.MemoryCache* method), 80
`duplicate_channel_counter_dict` (*antispam.dataclasses.member.Member* attribute), 73
`duplicate_counter` (*antispam.dataclasses.member.Member* attribute), 73
`DuplicateObject`, 69

E

`embed_to_string`() (*antispam.abc.Lib* method), 63
`embed_to_string`() (*antispam.libs.shared.base.Base* method), 87
`ENHANCED_DPY` (*antispam.enums.Library* attribute), 27
`ExistingEntry`, 69

G

`get_all_guilds`() (*antispam.abc.Cache* method), 60
`get_all_guilds`() (*antispam.caches.MemoryCache* method), 80
`get_all_members`() (*antispam.abc.Cache* method), 60
`get_all_members`() (*antispam.caches.MemoryCache* method), 80

`get_channel_by_id`() (*antispam.abc.Lib* method), 64
`get_channel_from_message`() (*antispam.abc.Lib* method), 64
`get_channel_id`() (*antispam.abc.Lib* method), 64
`get_file`() (*antispam.abc.Lib* method), 64
`get_guild`() (*antispam.abc.Cache* method), 60
`get_guild`() (*antispam.caches.MemoryCache* method), 80
`get_guild_data`() (*antispam.PluginCache* method), 43
`get_guild_id`() (*antispam.abc.Lib* method), 64
`get_guild_options`() (*antispam.AntiSpamHandler* method), 5
`get_member`() (*antispam.abc.Cache* method), 60
`get_member`() (*antispam.caches.MemoryCache* method), 80
`get_member_data`() (*antispam.PluginCache* method), 43
`get_member_from_message`() (*antispam.abc.Lib* method), 64
`get_message_mentions`() (*antispam.abc.Lib* method), 64
`get_options`() (*antispam.AntiSpamHandler* method), 5
`get_substitute_args`() (*antispam.abc.Lib* method), 64
`get_substitute_args`() (*antispam.libs.shared.base.Base* method), 87
`GUILD` (*antispam.enums.IgnoreType* attribute), 27
`Guild` (class in *antispam.dataclasses.guild*), 71
`guild_icon` (*antispam.libs.shared.substitute_args.SubstituteArgs* attribute), 89
`guild_id` (*antispam.dataclasses.member.Member* attribute), 73
`guild_id` (*antispam.dataclasses.message.Message* attribute), 75
`guild_id` (*antispam.dataclasses.propagate_data.PropagateData* attribute), 85
`guild_id` (*antispam.libs.shared.substitute_args.SubstituteArgs* attribute), 89
`guild_id` (*antispam.plugins.MassMentionPunishment* attribute), 49
`guild_log_ban_message` (*antispam.dataclasses.options.Options* attribute), 35
`guild_log_ban_message_delete_after` (*antispam.dataclasses.options.Options* attribute), 35
`guild_log_kick_message` (*antispam.dataclasses.options.Options* attribute), 35
`guild_log_kick_message_delete_after` (*antispam.dataclasses.options.Options* attribute), 35
`guild_log_timeout_message` (*antispam.dataclasses.options.Options* attribute),

- 35
 guild_log_timeout_message_delete_after (anti-spam.dataclasses.options.Options attribute), 35
 guild_log_warn_message (anti-spam.dataclasses.options.Options attribute), 35
 guild_log_warn_message_delete_after (anti-spam.dataclasses.options.Options attribute), 35
 guild_name (antispam.libs.shared.substitute_args.SubstituteArgs attribute), 89
 GuildAddonNotFound, 69
 GuildNotFound, 69
- ## H
- has_perms_to_make_guild (anti-spam.dataclasses.propagate_data.PropagateData attribute), 85
 HIKARI (antispam.enums.Library attribute), 27
- ## I
- id (antispam.dataclasses.guild.Guild attribute), 71
 id (antispam.dataclasses.member.Member attribute), 73
 id (antispam.dataclasses.message.Message attribute), 75
 ignore_bots (antispam.dataclasses.options.Options attribute), 35
 ignored_channels (anti-spam.dataclasses.options.Options attribute), 35
 ignored_guilds (antispam.dataclasses.options.Options attribute), 35
 ignored_members (anti-spam.dataclasses.options.Options attribute), 35
 ignored_roles (antispam.dataclasses.options.Options attribute), 35
 IgnoreType (class in antispam.enums), 27
 init() (antispam.AntiSpamHandler method), 5
 initialize() (antispam.abc.Cache method), 60
 initialize() (antispam.caches.MemoryCache method), 80
 injectable_nonce (antispam.plugins.Stats attribute), 51
 internal_is_in_guild (anti-spam.dataclasses.member.Member attribute), 73
 InvalidMessage, 69
 InvocationCancelled, 69
 is_dm() (antispam.abc.Lib method), 64
 is_duplicate (antispam.dataclasses.message.Message attribute), 75
 is_member_currently_timed_out() (anti-spam.abc.Lib method), 64
 is_overall_punishment (anti-spam.plugins.MassMentionPunishment attribute), 49
 is_per_channel_per_guild (anti-spam.dataclasses.options.Options attribute), 35
- ## K
- kick_count (antispam.dataclasses.member.Member attribute), 73
 KICKS_COUNTER (antispam.enums.ResetType attribute), 27
 kick_threshold (antispam.dataclasses.options.Options attribute), 35
- ## L
- Lib (class in antispam.abc), 61
 lib_embed_as_dict() (antispam.abc.Lib method), 65
 lib_embed_as_dict() (anti-spam.libs.shared.base.Base method), 87
 Library (class in antispam.enums), 27
 load_from_dict() (antispam.AntiSpamHandler static method), 5
 load_from_dict() (antispam.BasePlugin class method), 91
 load_from_dict() (antispam.plugins.Stats class method), 51
 log_channel_id (antispam.dataclasses.guild.Guild attribute), 71
 LogicError, 69
- ## M
- MassMentionPunishment (class in antispam.plugins), 49
 MaxMessageLimiter (class in antispam.plugins), 55
 MEMBER (antispam.enums.IgnoreType attribute), 27
 Member (class in antispam.dataclasses.member), 73
 member_avatar (antispam.libs.shared.substitute_args.SubstituteArgs attribute), 89
 member_ban_message (anti-spam.dataclasses.options.Options attribute), 35
 member_ban_message_delete_after (anti-spam.dataclasses.options.Options attribute), 35
 member_duplicate_count (antispam.CorePayload attribute), 37
 member_failed_ban_message (anti-spam.dataclasses.options.Options attribute), 35
 member_failed_kick_message (anti-spam.dataclasses.options.Options attribute), 35

`member_failed_timeout_message` (*antispyam.dataclasses.options.Options* attribute), 35
`member_id` (*antispyam.dataclasses.propagate_data.PropagateData* attribute), 85
`member_id` (*antispyam.libs.shared.substitute_args.SubstituteArgs* attribute), 89
`member_id` (*antispyam.plugins.MassMentionPunishment* attribute), 49
`member_kick_count` (*antispyam.CorePayload* attribute), 37
`member_kick_message` (*antispyam.dataclasses.options.Options* attribute), 35
`member_kick_message_delete_after` (*antispyam.dataclasses.options.Options* attribute), 35
`member_name` (*antispyam.dataclasses.propagate_data.PropagateData* attribute), 85
`member_name` (*antispyam.libs.shared.substitute_args.SubstituteArgs* attribute), 89
`member_should_be_punished_this_message` (*antispyam.CorePayload* attribute), 37
`member_status` (*antispyam.CorePayload* attribute), 38
`member_timeout_message` (*antispyam.dataclasses.options.Options* attribute), 35
`member_timeout_message_delete_after` (*antispyam.dataclasses.options.Options* attribute), 35
`member_warn_count` (*antispyam.CorePayload* attribute), 38
`member_warn_message` (*antispyam.dataclasses.options.Options* attribute), 35
`member_warn_message_delete_after` (*antispyam.dataclasses.options.Options* attribute), 35
`member_was_banned` (*antispyam.CorePayload* attribute), 38
`member_was_kicked` (*antispyam.CorePayload* attribute), 38
`member_was_timed_out` (*antispyam.CorePayload* attribute), 38
`member_was_warned` (*antispyam.CorePayload* attribute), 38
`MemberAddonNotFound`, 69
`MemberNotFound`, 70
`members` (*antispyam.dataclasses.guild.Guild* attribute), 71
`MemoryCache` (class in *antispyam.caches*), 79
`mention_bot` (*antispyam.libs.shared.substitute_args.SubstituteArgs* property), 89
`mention_member` (*antispyam.libs.shared.substitute_args.SubstituteArgs* property), 89
`mention_on_embed` (*antispyam.dataclasses.options.Options* attribute), 35
`Message` (class in *antispyam.dataclasses.message*), 75
`message_duplicate_accuracy` (*antispyam.dataclasses.options.Options* attribute), 35
`message_duplicate_count` (*antispyam.dataclasses.options.Options* attribute), 35
`message_interval` (*antispyam.dataclasses.options.Options* attribute), 35
`messages` (*antispyam.dataclasses.guild.Guild* attribute), 71
`messages` (*antispyam.dataclasses.member.Member* attribute), 71
`MissingGuildPermissions`, 70
`module`
`antispyam.exceptions`, 69

N

`NEXTCORD` (*antispyam.enums.Library* attribute), 27
`no_punish` (*antispyam.dataclasses.options.Options* attribute), 35
`NonExistentEntry`, 70
`NotFound`, 70

O

`ObjectMismatch`, 70
`options` (*antispyam.dataclasses.guild.Guild* attribute), 71
`Options` (class in *antispyam.dataclasses.options*), 29

P

`per_channel_spam` (*antispyam.dataclasses.options.Options* attribute), 35
`PINCER` (*antispyam.enums.Library* attribute), 27
`PluginCache` (class in *antispyam*), 43
`PluginError`, 70
`pre_invoke_extensions` (*antispyam.CorePayload* attribute), 38
`propagate()` (*antispyam.AntiSpamHandler* method), 6
`propagate()` (*antispyam.BasePlugin* method), 91
`propagate()` (*antispyam.plugins.AdminLogs* method), 53
`propagate()` (*antispyam.plugins.AntiMassMention* method), 50
`propagate()` (*antispyam.plugins.MaxMessageLimiter* method), 55
`propagate()` (*antispyam.plugins.Stats* method), 51
`PropagateData` (class in *antispyam.dataclasses.propagate_data*), 85
`PropagateFailure`, 70

`punish_member()` (*antispam.abc.Lib method*), 65
`PYCORD` (*antispam.enums.Library attribute*), 27

R

`register_plugin()` (*antispam.AntiSpamHandler method*), 6
`remove_guild_log_channel()` (*antispam.AntiSpamHandler method*), 7
`remove_guild_options()` (*antispam.AntiSpamHandler method*), 7
`remove_ignored_item()` (*antispam.AntiSpamHandler method*), 7
`reset_member_count()` (*antispam.abc.Cache method*), 60
`reset_member_count()` (*antispam.AntiSpamHandler method*), 7
`reset_member_count()` (*antispam.caches.MemoryCache method*), 80
`ResetType` (*class in antispam.enums*), 27
`ROLE` (*antispam.enums.IgnoreType attribute*), 27

S

`save_to_dict()` (*antispam.AntiSpamHandler method*), 8
`save_to_dict()` (*antispam.BasePlugin method*), 91
`save_to_dict()` (*antispam.plugins.Stats method*), 52
`send_guild_log()` (*antispam.abc.Lib method*), 65
`send_message_to_()` (*antispam.abc.Lib method*), 66
`set_cache()` (*antispam.AntiSpamHandler method*), 8
`set_guild()` (*antispam.abc.Cache method*), 61
`set_guild()` (*antispam.caches.MemoryCache method*), 81
`set_guild_data()` (*antispam.PluginCache method*), 43
`set_member()` (*antispam.abc.Cache method*), 61
`set_member()` (*antispam.caches.MemoryCache method*), 81
`set_member_data()` (*antispam.PluginCache method*), 44
`Stats` (*class in antispam.plugins*), 51
`substitute_args()` (*antispam.abc.Lib method*), 66
`substitute_args()` (*antispam.libs.shared.base.Base method*), 87
`SubstituteArgs` (*class in antispam.libs.shared.substitute_args*), 89

T

`timeout_member()` (*antispam.abc.Lib method*), 67
`times_timed_out` (*antispam.dataclasses.member.Member attribute*), 73
`timestamp_now` (*antispam.libs.shared.substitute_args.SubstituteArgs property*), 89

`timestamp_today` (*antispam.libs.shared.substitute_args.SubstituteArgs property*), 89
`transform_message()` (*antispam.abc.Lib method*), 67
`transform_message()` (*antispam.libs.shared.base.Base method*), 88

U

`unregister_plugin()` (*antispam.AntiSpamHandler method*), 8
`UnsupportedAction`, 70
`use_timeouts` (*antispam.dataclasses.options.Options attribute*), 35

V

`visualize()` (*antispam.AntiSpamHandler method*), 8
`visualizer()` (*antispam.abc.Lib method*), 67
`visualizer()` (*antispam.libs.shared.base.Base method*), 88

W

`warn_count` (*antispam.dataclasses.member.Member attribute*), 73
`WARN_COUNTER` (*antispam.enums.ResetType attribute*), 27
`warn_only` (*antispam.dataclasses.options.Options attribute*), 35
`warn_threshold` (*antispam.dataclasses.options.Options attribute*), 36