Discord Advertisement Framework

David Hozic

Mar 25, 2023

CONTENTS

The Discord advertisement framework is a Python based **shilling framework** that allows easy advertising on Discord.

CHAPTER

ONE

LINKS

Project

- Github
- Examples
- Releases

API Wrapper (Pycord)

This framework uses a Discord API wrapper called PyCord and it is built to allow working directly with Pycord (eg. framework objects accept Pycord objects as arguments).

- PyCord GitHub
- PyCord Documentation

CHAPTER

TWO

KEY FEATURES

- Ability to run on **multiple** accounts at once, either on a personal account or as **normal bot** account.

Caution: While running this on user accounts is possible, it is **not recommended** since it is against Discord's ToS. I am not responsible if your account get's disabled for using self-bots!

- Periodic advertisement to different channels,
- Login with username/password - *Automatic login*,
- Automatic guild discovery and join - *Automatic guild discovery and join*,
- Logging of sent messages (including SQL) - *Logging*
- Async framework
- Easy to setup, with minimal code

CHAPTER

THREE

INSTALLATION

DAF can be installed though command prompt/terminal using the bottom commands.

Main package

Pre-requirement: Python (minimum v3.10)

```
pip install discord-advert-framework
```

Additional functionality

Some functionality needs to be installed separately. This was done to reduce the needed space by the daf.

Voice

Listing 3.1: Voice Messaging / AUDIO

- `pip install discord-advert-framework[voice]`

Proxies

Listing 3.2: Proxy support

- `pip install discord-advert-framework[proxy]`

SQL

Listing 3.3: SQL logging

- `pip install discord-advert-framework[sql]`

All

Install all of the (left) optional dependencies

Listing 3.4: SQL logging

- `pip install discord-advert-framework[all]`

CHAPTER

FOUR

TABLE OF CONTENTS

4.1 Guide

This section contains the guide to using this framework.

4.1.1 Quickstart

This page contains information to quickly getting started.

The first thing you need is the library installed, see *Installation*.

Framework control

Only one function is needed to be called for the framework to start.

The framework can be started using *daf.core.run()* function (and stopped with the *daf.core.shutdown()* function).

Function *run()* accepts many parameters but there is only one that is most important:

accounts

Accounts parameter is a list of *daf.client.ACCOUNT* objects which represent different Discord accounts you can simultaneously use to shill your content.

The below example shows a minimum definition of the accounts list. For information about parameters with specific object, please use the search bar or refer to the *Programming Reference*.

See also:

To login with **username** and **password** instead of the account token, see *Automatic login*

See also:

The below example shows a bare minimum definition of the accounts list that has a **manually defined** server list.

There is also a way to automatically define the server list (and channels) based on the guild name (*Shilling scheme generation*).

Listing 4.1: Example

```
import daf

accounts = [
```

(continues on next page)

(continued from previous page)

```
    daf.client.ACCOUNT( # Account 1
        token="DJHADJHSKJDHAKHDSKJADHKASJ", # Account token
        is_user=False,  # Is the token from an user account?
        servers=[   # List of guilds/users
            daf.guild.GUILD(
                snowflake=123456789, # Snowflake id of discord
                messages=[
                    daf.message.TextMESSAGE(...),
                    daf.message.TextMESSAGE(...),
                    daf.message.VoiceMESSAGE(...)
                ],
                logging=True, # Log sent messages
                remove_after=None # To automatically stop shilling
            )
        ]
    ),

    daf.client.ACCOUNT( # Account 2
        token="JKDJSKDJALKNDSAKNDASKNDKAJS", # Account token
        is_user=False,  # Is the token from an user account?
        servers=[   # List of guilds/users
            daf.guild.GUILD(
                snowflake=123456789, # Snowflake id of discord
                messages=[
                    daf.message.TextMESSAGE(...),
                    daf.message.TextMESSAGE(...),
                    daf.message.VoiceMESSAGE(...)
                ],
                logging=True, # Log sent messages
                remove_after=None # To automatically stop shilling
            )
        ]
    )
]


daf.run(accounts=accounts)
```

After you've successfully defined your accounts list and started the framework with *run()*, the framework will run on it's own and there is nothing you need to do from this point forward if basic periodic shilling with text messages is all you desire.

4.1.2 Sending messages

This document holds information regarding shilling with message objects.

Guild types

Before you start sending any messages you need to define a *GUILD* / *USER*. object. The *GUILD* objects represents Discord servers with text/voice channels and it can hold *TextMESSAGE* and *VoiceMESSAGE* messages, while *USER* represents a single user on Discord and can hold *DirectMESSAGE* messages. For more information about how to use *GUILD* / *USER* click on them.

Guilds can be passed to the framework at startup (see *Quickstart*) and while the framework is running (see *Modifying the shilling list*).

Message types

Periodic messages can be represented with instances of **x**MESSAGE classes, where **x** represents the channel type. The channel logic is merged with the message logic which is why there are 3 message classes you can create instances from. These classes accept different parameters but still have some in common:

- `start_period` - If not None, represents bottom range of randomized period
- `end_period` - If `start_period` is not None, this represents upper range of randomized period, if `start_period` is None, represents fixed sending period.
- `data` (varies on message types) - data that is actually send to Discord.
- `start_in` - Defines when the message the shilling of message should stop (object be removed from framework).

For more information about these, see *TextMESSAGE*, *VoiceMESSAGE*, *DirectMESSAGE*.

Text messages

To periodically send text messages you'll have to use either *TextMESSAGE* for sending to text channels inside the guild or *DirectMESSAGE* for sending to user's private DM. To add these messages to the guild, set the *GUILD* / *USER*'s `messages` parameter to a table that has the message objects inside.

Voice messages

Shilling an audio message requires *VoiceMESSAGE* objects. You can only stream audio to guilds, users(direct messages) are not supported. You can either stream a fixed audio file or a youtube video, both thru *daf.dtypes.AUDIO* object.

4.1.3 Logging

The framework allows to log sent messages for each *GUILD*/*USER* (if you set the "logging" to True inside the *GUILD* or *USER* object).

Logging is handled thru so called **logging managers**. Currently, 3 different managers exists:

- LoggerJSON: Used for saving file logs in the JSON format. (*JSON Logging (file)*)
- LoggerCSV: Used for saving file logs in the CSV format, where certain fields are still JSON. (*CSV Logging (file)*)
- LoggerSQL: Used for saving relational database logs into a remote database. (*Relational Database Log (SQL)*)

- Custom logger: User can create a custom logger if they desire. (*Custom Logger*)

If a logging managers fails saving a log, then it's fallback manager will be used temporarily to store the log. It will only use the fallback once and then, at the next message, the original manager will be used.

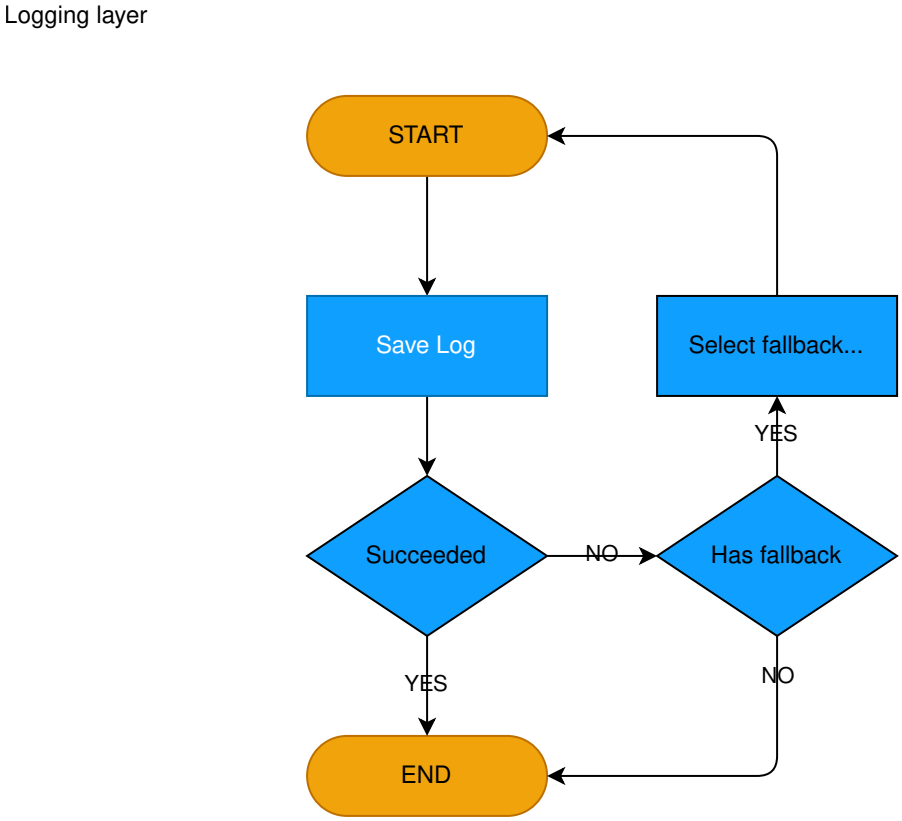


Fig. 4.1: Logging process with fallback

JSON Logging (file)

The logs are written in the JSON format and saved into a JSON file, that has the name of the guild or an user you were sending messages into. The JSON files are fragmented by day and stored into folder `Year/Month/Day`, this means that each day a new JSON file will be generated for that specific day for easier managing, for example, if today is `13.07.2022`, the log will be saved into the file that is located in

```
History
└──2022
|   └──07
|       └──13
|           └── #David's dungeon.json
```

JSON structure

The log structure is the same for both *USER* and *GUILD*. All logs will contain keys:

- "name": The name of the guild/user
- "id": Snowflake ID of the guild/user
- "type": object type (GUILD/USER) that generated the log.
- "message_history": Array of logs for each sent message to the guild/user, the structure is message type dependant and is generated inside methods:
  - *daf.message.TextMESSAGE.generate_log_context()*
  - *daf.message.VoiceMESSAGE.generate_log_context()*
  - *daf.message.DirectMESSAGE.generate_log_context()*

See also:

`Example structure`

CSV Logging (file)

The logs are written in the CSV format and saved into a CSV file, that has the name of the guild or an user you were sending messages into. The CSV files are fragmented by day and stored into folder `Year/Month/Day`, this means that each day a new CSV file will be generated for that specific day for easier managing, for example, if today is `13.07.2023`, the log will be saved into the file that is located in

```
History
└──2023
|   └──07
|       └──13
|           └── #David's dungeon.csv
```

CSV structure

The structure contains the following attributes:

- Timestamp (string)
- Guild Type (string),
- Guild Name (string),
- Guild Snowflake (integer),
- Message Type (string),
- Sent Data (json),
- Message Mode (non-empty for *TextMESSAGE* and *DirectMESSAGE*) (string),
- Message Channels (non-empty for *TextMESSAGE* and *VoiceMESSAGE*) (json),
- Success Info (non-empty for *DirectMESSAGE*) (json),

Note: Attributes marked with `(json)` are the same as in *JSON Logging (file)*

See also:

`Structure example`

Relational Database Log (SQL)

New in version v1.9.

Changed in version v2.1:

Turned into an optional feature.

```
pip install discord-advert-framework[sql]
```

Changed in version v2.2:

Additional dialect support
Microsoft SQL Server, PostgreSQL, MariaDB/MySQL, SQLite

Better Caching
Improved caching to significantly increase logging speed

asynchronous
All of the SQL connectors except MS SQL Server are asynchronous.

This type of logging enables saving logs to a remote server inside the database. In addition to being smaller in size, database logging takes up less space and it allows easier data analysis.

Dialects

The dialect is selected via the `dialect` parameter in *LoggerSQL*. The following dialects are supported:

- Microsoft SQL Server
- PostgreSQL
- SQLite,
- MySQL

Usage

For daf to use SQL logging, you need to pass the *run()* function with the `logger` parameter and pass it the *LoggerSQL* object.

Features

- Automatic creation of the schema
- Caching for faster logging
- Low redundancy for reduced file size
- Automatic error recovery

Note: The database must already exist! However it can be completely empty, no need to manually create the schema.

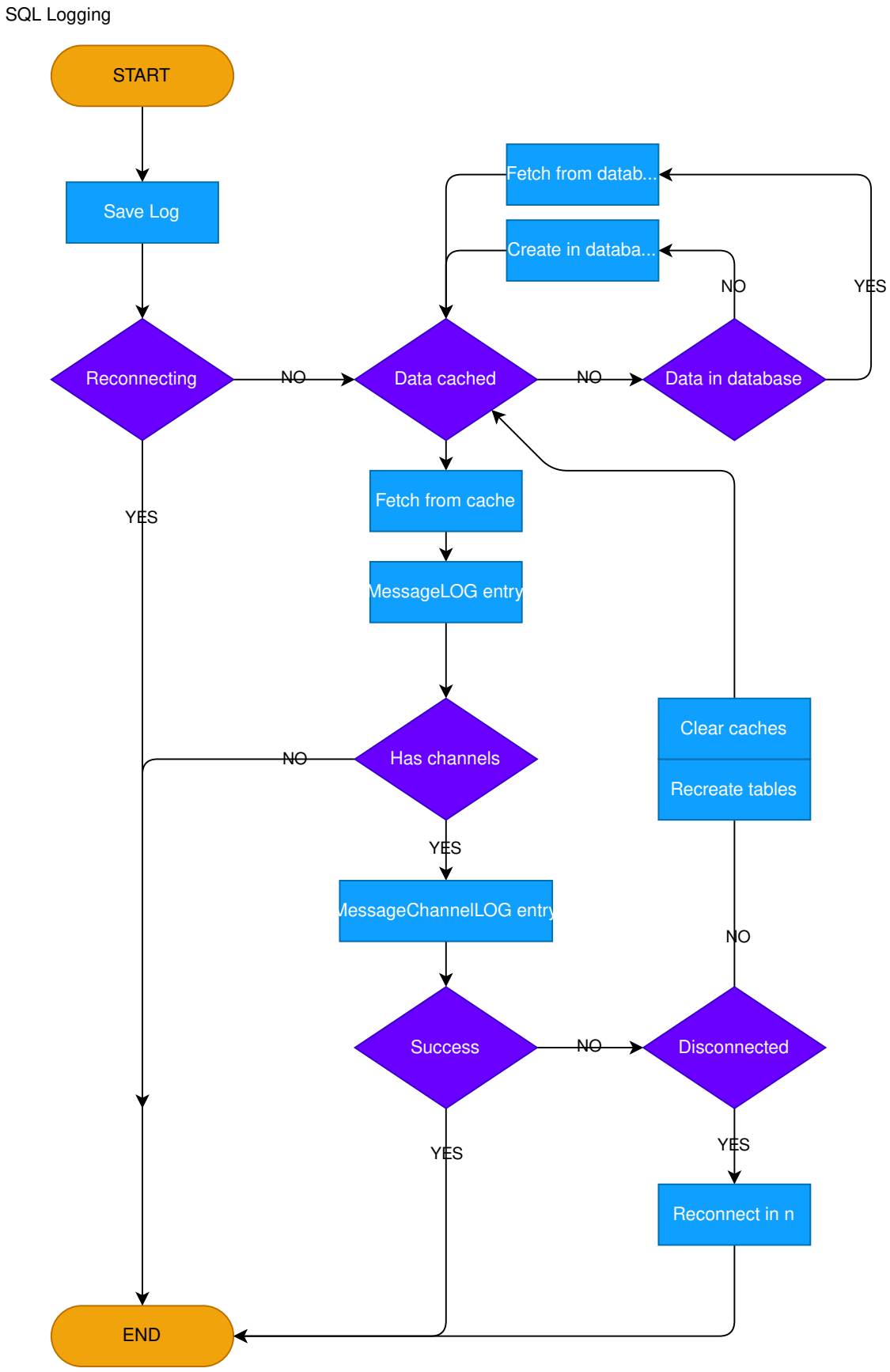
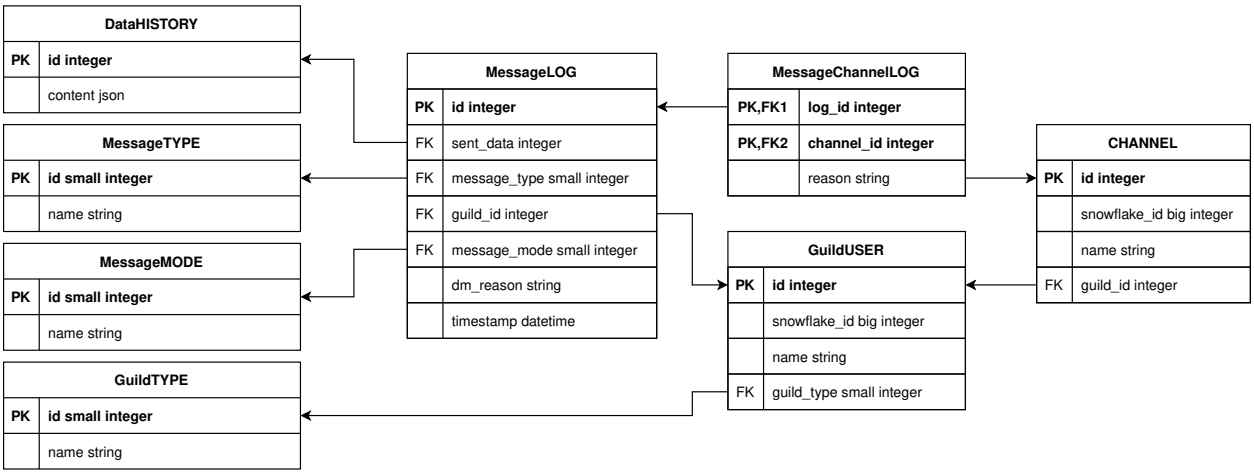


Fig. 4.2: SQL Logging diagram

ER diagram



Tables

MessageLOG

Description

This table contains the actual logs of sent messages, if the message type is *DirectMESSAGE*, then all the information is stored in this table. If the types are **Voice/Text** MESSAGE, then channel part of the log is saved in the *MessageChannelLOG* table.

Attributes

- **[Primary Key]** id: Integer - This is an internal ID of the log inside the database.
- sent_data: Integer - Foreign key pointing to a row inside the *DataHISTORY* table.
- message_type: SmallInteger - Foreign key ID pointing to a entry inside the *MessageTYPE* table.
- guild_id: Integer - Foreign key pointing to *GuildUSER* table.
- message_mode: SmallInteger - Foreign key pointing to *MessageMODE* table. This is non-null only for *DirectMESSAGE*.
- dm_reason: String - If MessageTYPE is not DirectMESSAGE or the send attempt was successful, this is NULL, otherwise it contains the string representation of the error that caused the message send attempt to be unsuccessful.
- timestamp: DateTime - The timestamp of the message send attempt.

DataHISTORY

Description

This table contains all the **different** data that was ever advertised. Every element is **unique** and is not replicated. This table exist to reduce redundancy and file size of the logs whenever same data is advertised multiple times. When a log is created, it is first checked if the data sent was already sent before, if it was the id to the existing *DataHISTORY* row is used, else a new row is created.

Attributes

- **[Primary Key]** id: Integer - Internal ID of data inside the database.
- content: JSON - Actual data that was sent.

MessageTYPE

Description

This is a lookup table containing the the different message types that exist within the framework (*Messages*).

Attributes

- **[Primary Key]** id: SmallInteger - Internal ID of the message type inside the database.
- name: String - The name of the actual message type.

GuildUSER

Description

The table contains all the guilds/users the framework ever generated a log for.

Attributes

- **[Primary Key]** id: Integer - Internal ID of the Guild/User inside the database.
- snowflake_id: BigInteger - The discord (snowflake) ID of the User/Guild
- name: String - Name of the Guild/User
- guild_type: SmallInteger - Foreign key pointing to *GuildTYPE* table.

MessageMODE

Description

This is a lookup table containing the the different message modes available by *TextMESSAGE* / *DirectMESSAGE*, it is set to null for *VoiceMESSAGE*.

Attributes

- **[Primary Key]** id: SmallInteger - Internal identifier of the message mode inside the database.
- name: String - The name of the actual message mode.

GuildTYPE

Description
This is a lookup table containing types of the guilds inside the framework (*Guilds*).

Attributes

- **[Primary Key]** id: SmallInteger - Internal identifier of the guild type inside the database.
- name: String - The name of the guild type.

CHANNEL

Description
The table contains all the channels that the framework ever advertised into.

Attributes

- **[Primary Key]** id: Integer - Internal identifier of the channel inside the database
- snowflake_id: BigInteger - The discord (snowflake) identifier representing specific channel
- name: String - The name of the channel
- guild_id: Integer - Foreign key pointing to a row inside the *GuildUSER* table. It points to a guild that the channel is part of.

MessageChannelLOG

Description
Since messages can send into multiple channels, each MessageLOG has multiple channels which cannot be stored inside the *MessageLOG*. This is why this table exists. It contains channels of each *MessageLOG*.

Attributes

- **[Primary Key] [Foreign Key]** log_id: Integer - Foreign key pointing to a row inside *MessageLOG* (to which log this channel log belongs to).
- **[Primary Key] [Foreign Key]** channel_id: Integer - Foreign key pointing to a row inside the *CHANNEL* table.
- reason: String - Reason why the send failed or `NULL` if send succeeded.

Custom Logger

If you want to use a different logging scheme than the ones built in, you can do so by creating a custom logging manager that inherits the *daf.logging.LoggerBASE*.

The derived logger class can then implement the following methods:

1. **__init__(self, param1, param2, ...) [Required]:**
   The method used for passing parameters and for basic non-async initialization. This method must contain a fallback parameter and also needs to have an attribute of the same name.

Listing 4.2: Custom __init__ method

```
class LoggerCUSTOM(daf.logging.LoggerBASE):
    def __init__(self, ..., logger):
        ... # Set attributes
        super().__init__(logger)

    ... # Other methods
```

2. **async initialize(self) [Optional]:**
   The base's `initialize` method calls `initialize` method of it's fallback, if it fails then the fallback is set to None.

   If you wish to do additional initialization that requires async/await operations, you can implement your own `initialize` method but make sure you call the base's method in the end.

Listing 4.3: Custom initialize method

```
class LoggerCUSTOM(daf.logging.LoggerBASE):
    ... # Other methods

    async def initialize(self):
        ... # Custom implementation code
        await super().initialize()
```

3. **async _save_log(self, guild_context: dict, message_context: dict) [Required]:**
   Method that stores the message log. If there is any error in saving the log an exception should be raised, which will then make the logging module automatically use the fallback manager, **do not call the fallback manager from this method!**

   Parameters

   guild_context (dict) - Contains keys:

   - "name": The name of the guild/user (str)
   - "id": Snowflake ID of the guild/user (int)
   - "type": object type (GUILD/USER) that generated the log. (str)

   message_context (dict) - Dictionary returned by:

   - *daf.message.TextMESSAGE.generate_log_context()*
   - *daf.message.VoiceMESSAGE.generate_log_context()*
   - *daf.message.DirectMESSAGE.generate_log_context()*

4. **async update(self, **kwargs) [Optional]:**
   Custom implementation of the `update` method.

   This method is used for updating the parameters that are available thru `__init__` method and **is not required if the attributes inside the object have the same name as the parameters inside the `__init__` function** and there are no pre-required steps that need to be taken before updating (see *JSON Logging (file)*'s code for example).

   However if the name of attributes differ from parameter name or the attribute doesn't exist at all or other steps are required than just re-initialization (see *daf.logging.sql.LoggerSQL*'s update method), then this method is required to be implemented. It should be implemented in a way that it calls the base update method. Example:

```
class LoggerCUSTOM(daf.logging.LoggerBASE):
    def __init__(self, name, fallback):
        self._name = name
        super().__init__(fallback)

    ... # Other methods

    async def update(self, **kwargs)
        # Only modify if the parameter is not passed to update method
        if "name" not kwargs:
            # The name parameter is stored under "_name" attribute instead of
↪"name"
            kwargs["name"] = self._name

        ... # Other pre-required code (eg. remote SQL server needs to be␣
↪disconnected)

        super().update(**kwargs) # Call base update method
```

4.1.4 Dynamic modification

This document describes how the framework can be modified dynamically.

Modifying the shilling list

See *Dynamic mod.* for more information about the **functions** mentioned below.

While the shilling list can be defined statically (pre-defined) by creating a list and using the `servers` parameter in the *ACCOUNT* instances (see *Quickstart*), the framework also allows the objects to be added or removed dynamically from the user's program after the framework has already been started and initialized.

Dynamically adding objects

Objects can be dynamically added using the *daf.core.add_object()* coroutine function. The function can be used to add the following object types:

Accounts

daf.client.ACCOUNT

Guilds

daf.guild.GUILD

daf.guild.USER

daf.guild.AutoGUILD

Messages

daf.message.TextMESSAGE

daf.message.VoiceMESSAGE

daf.message.DirectMESSAGE

Note: Messages can also be added thru the *daf.guild.GUILD.add_message()* / *daf.guild.USER.add_message()* method.

Caution: The guild must already be added to the framework, otherwise this method will fail.

```
...
my_guild = daf.GUILD(guild.id, logging=True)
await daf.add_object(my_guild, account)
await my_guild.add_message(daf.TextMESSAGE(...))
...
```

Dynamically removing objects

As the framework supports dynamically adding new objects to the shilling list, it also supports dynamically removing those objects. Objects can be removed with the *daf.core.remove_object()*.

Modifying objects

Some objects in the framework can be dynamically updated thru the `.update()` method. The principle is the same for all objects that support this and what this method does is it updates the original parameters that can be passed during object creation.

Warning: This completely resets the state of the object you are updating, meaning that if you do call the `.update()` method, the object will act like it was recreated.

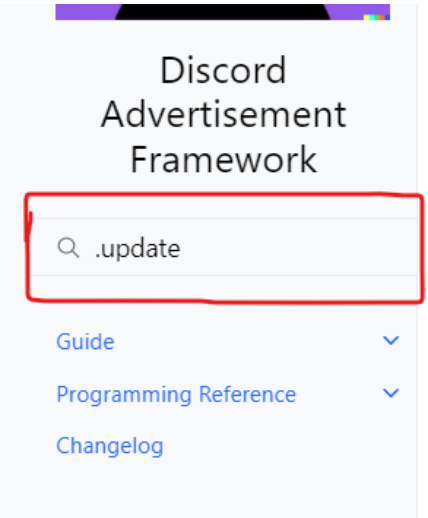
For example if I wanted to change the shilling period of a *daf.message.TextMESSAGE*, I would call the *daf.message.TextMESSAGE.update()* method in the following way:

```
... # Other code
# Fixed sending period of 5 seconds
my_message = daf.message.TextMESSAGE(
                                        start_period=None,
                                        end_period=timedelta(seconds=5),
                                        ... # Other parameters
                                    )


await daf.add_object(my_message, some_GUILD_object)

# Randomized sending period between 3 and 5 seconds
await my_message.update(start_period=timedelta(seconds=3))
... # Other code
```

For a full list of objects that support `.update` search ".update" in the search bar **or click on the image below**.



daf.logging.LoggerBASE.update (Python method, in Classes)

daf.logging.sql.LoggerSQL.update (Python method, in Classes)

daf.message.DirectMESSAGE.update (Python method, in Classes)

daf.message.TextMESSAGE.update (Python method, in Classes)

daf.message.VoiceMESSAGE.update (Python method, in Classes)

4.1.5 Automatic generation

This documents describes mechanisms that can be used to automatically generate objects.

Shilling scheme generation

While the framework supports to manually define a schema, which can be time consuming if you have a lot of guilds to shill into and harder to manage, the framework also supports automatic generation of the schema.

Using this method allows you to have a completely automatically managed system of finding guilds and channels that match a specific regex pattern. It automatically finds new guilds/channels at initialization and also during normal framework operation. This is great because it means you don't have to do much but it gives very little control of into what to shill.

Automatic GUILD generation

See also:

This section only describes guilds that the user **is already joined in**. For information about **discovering NEW guilds and automatically joining them** see *Automatic guild discovery and join*

For a auto-managed GUILD list, use *AutoGUILD* which internally generates *GUILD* instances. Simply create a list of *AutoGUILD* objects and then pass it to the framework. It can be passed to the framework exactly the same way as *GUILD* (see *Quickstart* (`accounts`) and *Dynamically adding objects*).

Warning: Messages that are added to *AutoGUILD* should have *AutoCHANNEL* for the `channels` parameters, otherwise you will be spammed with warnings and only one guild will be shilled.

```
from datetime import timedelta
import daf


# STATIC DEFINITION
```

(continues on next page)

(continued from previous page)

```
ACCOUNTS = [
    daf.ACCOUNT(
        token="SomeToken",
        is_user=False,
        servers=[
            daf.guild.AutoGUILD( include_pattern="NFT-Dragons", # Regex pattern of guild␣
↪names that should be included
                                 exclude_pattern="NonFunctionalTesticle",   # Regex␣
↪pattern of guild names that should be excluded
                                 remove_after=None, # Never stop automatically managing␣
↪guilds
                                 messages=[
                                    daf.message.TextMESSAGE(None, timedelta(seconds=5),
↪"Buy our NFT today!", daf.message.AutoCHANNEL("shill", "promo", timedelta(seconds=60)))
                                 ],
                                 logging=True, # The generated GUILD objects will have␣
↪logging enabled
                                 interval=timedelta(hours=1) ) # Scan for new guilds in␣
↪a period of one hour
        ]
    )
]



daf.run(
    accounts=ACCOUNTS
)
```

Automatic channel generation

For a auto-managed channel list use *AutoCHANNEL* instances. It can be passed to xMESSAGE objects into the `channels` parameters instead of a list.

Listing 4.4: AutoCHANNEL example

```
from datetime import timedelta
import daf


async def main():
    await daf.add_object(
        daf.ACCOUNT(
            token="SomeToken",
            is_user=False,
            servers=[
                daf.GUILD(snowflake=123456789,
                          messages=[
                                daf.TextMESSAGE(None, timedelta(seconds=5), "Hello World
↪", channels=daf.message.AutoCHANNEL("shill", exclude_pattern="shill-[7-9]"))
                             ],
```

(continues on next page)

(continued from previous page)

```
                            logging=True)
            ]
        )
    )


daf.run(
    user_callback=main,
)
```

4.1.6 Web browser

Warning: This can only be used if you are running the framework in a desktop environment. You cannot use it eg. on a Ubuntu server.

DAF includes optional functionality that allows automatic guild joins and login with username and password instead of token.

To use the web functionality, users need to install the optional packages with:

```
pip install discord-advert-framework[web]
```

The Chrome browser is also required to run the web functionality.

Note: When running the web functionality, the `proxy` parameter passed to *ACCOUNT*, will also be used to the browser.

Unfortunetly it is not directly possible for the web driver to accept a proxy with username and password, meaning just the normal "protocol://ip:port" proxy will work. If you plan to run a private proxy, it is recommened that you whitelist your IP instead and pass the `proxy` paramterer in the "protocol://ip:port" format.

Automatic login

To login with username (email) and password instead of the token, users need to provide the *ACCOUNT* object with the `username` and `password` parameters.

```
import daf

accounts = [
    daf.ACCOUNT(
        username="myemail@gmail.com",
        password="TheRiverIsFlowingDownTheHill232",
        ...
    )
]


daf.run(accounts=accounts)
```

If you run the above snippet, DAF will first open the browser, load the Discord login screen and login, then it will save the token into a file under "daf_web_data" folder and send the token back to the framework. The framework will then run exactly the same as it would if you passed it the token directly.

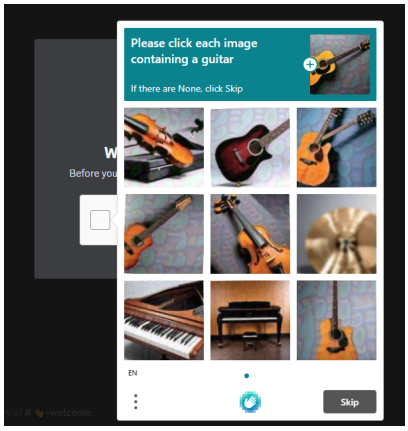
If you restart DAF, it will not re-login, but will just load the data from the saved storage under "daf_web_data" folder.

Automatic guild discovery and join

The web layer beside login with username and password, also allows (semi) automatic guild discovery and join.

To use this feature, users need to create an *AutoGUILD* instance, where they pass the `auto_join` parameter. `auto_join` parameter is a *GuildDISCOVERY* object, which can be configured how it should search for new guilds.

Warning: When joining a guild, users may be prompted to complete the **CAPTCHA** (Completely Automated Public Turing Check to tell Computers and Humans Apart), which is why this is **semi**-automatic. In the case of this event, the browser will wait 90 seconds for the user to complete the CAPTCHA, if they don't it will consider the join to be a failure.




```
from daf import QuerySortBy, QueryMembers
import daf

accounts = [
    daf.ACCOUNT(
        username="myemail@gmail.com",
        password="TheRiverIsFlowingDownTheHill232",
        servers=[
            daf.AutoGUILD(
                ".*",
                auto_join=daf.GuildDISCOVERY("NFT art", daf.QuerySortBy.TOP, limit=5),
                ...
            )
        ]
    )
]


daf.run(accounts=accounts)
```

With the above example, *AutoGUILD*, will search for guilds that match the `prompt` parameter and select the ones that

match the other parameters. After finding a guild, it will then check if the `include_pattern` parameter of *AutoGUILD* matches with the guild name and if it does, it will then obtain the invite link and try to join the guild.

The browser will only try to join as many guilds as defined by the `limit` parameter of *GuildDISCOVERY*. Guilds that the user is already joined, also count as a successful join, meaning that if the limit is eg. 5 and the users is joined into 3 of the found guilds, browser will only join 2 new guilds.

Web feature example

The following shows an example of both previously described features.

```
"""
This example shows how the user can use username and password to login into Discord and
it also shows how to configure the automatic guild discovery and join feature (auto_join
↪parameter)
"""

from daf import QuerySortBy, QueryMembers
import daf

accounts = [
    daf.ACCOUNT(
        username="email@gmail.com",
        password="Password6745;*",
        servers=[
            daf.AutoGUILD(
                include_pattern=".*",
                auto_join=daf.GuildDISCOVERY(prompt="NFT arts",
                                             sort_by=QuerySortBy.TOP,
                                             total_members=QueryMembers.ALL,
                                             limit=20),
            ),
        ],
        proxy="protocol://ip:port"
    )
]


daf.run(
    accounts=accounts,
    debug=daf.TraceLEVELS.NORMAL
)
```

4.2 Programming Reference

Contain classes and functions description.

4.2.1 Logging reference

trace

daf.logging.tracing.**trace**(*message: str*, *level: Union[*TraceLEVELS*, int] = TraceLEVELS.NORMAL*, *reason: Optional[Exception] = None*)

Prints a trace to the console. This is thread safe.

Changed in version v2.3:

Will only print if the level is lower than the configured (thru *run()*'s debug parameter max level.

Eg. if the max level is *ERROR*, then the level parameter needs to be either *DEPRECATED* or *ERROR*, else nothing will be printed.

Parameters

- **message** (*str*) – Trace message.
- **level** (*TraceLEVELS | int*) – Level of the trace. Defaults to TraceLEVELS.NORMAL.
- **reason** (*Optional[Exception]*) – Optional exception object, which caused the prinout.

get_logger

daf.logging.**get_logger**() → *LoggerBASE*

Returns
The selected logging object which is of inherited type from LoggerBASE.

Return type
LoggerBASE

TraceLEVELS

enum daf.logging.tracing.**TraceLEVELS**(*value*)

Levels of trace for debug.

See also:

trace

Changed in version v2.3: Added DEPRECATION

Member Type
int

Valid values are as follows:

DEPRECATED = <TraceLEVELS.DEPRECATED: 0>
Show only deprecation notices.

ERROR = <TraceLEVELS.ERROR: 1>

Show deprecations and errors.

WARNING = <TraceLEVELS.WARNING: 2>

Show deprecations, errors, warnings.

NORMAL = <TraceLEVELS.NORMAL: 3>

Show deprecations, errors, warnings, info messages.

DEBUG = <TraceLEVELS.DEBUG: 4>

Show deprecations, errors, warnings, info messages, debug messages.

LoggerBASE

class daf.logging.**LoggerBASE**(*fallback=None*)

New in version v2.2.

The base class for making loggers. This can be used to implement your custom logger as well. This does absolutely nothing, and is here just for demonstration.

Parameters

fallback (*Optional[LoggerBASE]*) – The manager to use, in case saving using this manager fails.

async initialize() → None

Initializes self and the fallback

async update(***kwargs*)

Used to update the original parameters.

Parameters

kwargs (*Any*) – Keyword arguments of any original parameters.

Raises

- **TypeError** – Invalid keyword argument was passed.
- **Other** – Other exceptions raised from `.initialize` method (if it exists).

LoggerCSV

class daf.logging.**LoggerCSV**(*path: str*, *delimiter: str*, *fallback: Optional[LoggerBASE] = None*)

New in version v2.2.

Logging class for generating .csv file logs. The logs are saved into CSV files and fragmented by guild/user and day (each day, new file for each guild).

Each entry is in the following format:

```
Timestamp, Guild Type, Guild Name, Guild Snowflake, Message Type, Sent Data, Message Mode (Optional), Channels (Optional), Success Info (Optional)
```

Parameters

- **path** (*str*) – Path to the folder where logs will be saved.
- **delimiter** (*str*) – The delimiter between columns to use.
- **fallback** (*Optional[LoggerBASE]*) – The manager to use, in case saving using this manager fails.

Raises
OSError – Something went wrong at OS level (insufficient permissions?) and fallback failed as well.

async initialize() → None

Initializes self and the fallback

async update(**kwargs)

Used to update the original parameters.

Parameters
kwargs (*Any*) – Keyword arguments of any original parameters.

Raises

- TypeError – Invalid keyword argument was passed.
- Other – Other exceptions raised from `.initialize` method (if it exists).

LoggerJSON

class daf.logging.LoggerJSON(*path: str*, *fallback: Optional[*LoggerBASE*] = None*)

New in version v2.2.

Logging class for generating .json file logs. The logs are saved into JSON files and fragmented by guild/user and day (each day, new file for each guild).

Parameters

- **path** (*str*) – Path to the folder where logs will be saved.
- **fallback** (*Optional[LoggerBASE]*) – The manager to use, in case saving using this manager fails.

Raises
OSError – Something went wrong at OS level (insufficient permissions?) and fallback failed as well.

async initialize() → None

Initializes self and the fallback

async update(**kwargs)

Used to update the original parameters.

Parameters
kwargs (*Any*) – Keyword arguments of any original parameters.

Raises

- TypeError – Invalid keyword argument was passed.
- Other – Other exceptions raised from `.initialize` method (if it exists).

LoggerSQL

class daf.logging.sql.**LoggerSQL**(*username: Optional[str] = None*, *password: Optional[str] = None*, *server: Optional[str] = None*, *port: Optional[int] = None*, *database: Optional[str] = None*, *dialect: Optional[str] = None*, *fallback: Optional[*LoggerBASE*] = Ellipsis*)

Used for controlling the SQL database used for message logs.

Parameters

- **username** (*Optional[str]*) – Username to login to the database with.
- **password** (*Optional[str]*) – Password to use when logging into the database.
- **server** (*Optional[str]*) – Address of the server.
- **port** (*Optional[int]*) – The port of the database server.
- **database** (*Optional[str]*) – Name of the database used for logs.
- **dialect** (*Optional[str]*) – Dialect or database type (SQLite, mssql,)
- **fallback** (*Optional[LoggerBASE]*) – The fallback manager to use in case SQL logging fails. (Default: *LoggerJSON* ("History"))

Raises

ValueError – Unsupported dialect (db type).

async **initialize**() → None

This method initializes the connection to the database, creates the missing tables and fills the lookup tables with types defined by the register_type(lookup_table) function.

Note: This is automatically called when running the daf.

Raises

Any – from ._begin_engine() from ._create_tables() from ._generate_lookup_values()

async **update**(***kwargs*)

New in version v2.0.

Used for changing the initialization parameters the object was initialized with.

Warning: Upon updating, the internal state of objects get's reset, meaning you basically have a brand new created object. It also resets the message objects.

Parameters

****kwargs** (*Any*) – Custom number of keyword parameters which you want to update, these can be anything that is available during the object creation.

Raises

- **TypeError** – Invalid keyword argument was passed.
- **Other** – Raised from .initialize() method.

4.2.2 Message data types

data_function

daf.dtypes.**data_function**(*fnc: Callable*)

Decorator used for wrapping a function that will return data to send when the message is ready.

The fnc function must return data that is of type that the **x**MESSAGE object supports. **If the type returned is not valid, the send attempt will simply be ignored and nothing will be logged at at**, this is useful if you want to use the fnc function to control whenever the message is ready to be sent. For example: if we have a function defined like this:

```
@daf.data_function
def get_data():
    return None

...
daf.TextMESSAGE(..., data=get_data())
...
```

then no messages will ever be sent, nor will any logs be made since invalid values are simply ignored by the framework.

Parameters
fnc (*Callable*) – The function to wrap.

Returns
A class for creating wrapper objects is returned. These wrapper objects can be used as a data parameter to the *Messages* objects.

Return type
FunctionCLASS

```
from datetime import timedelta
import  daf, datetime
from daf import discord



################################################################################
↪########
# It's VERY IMPORTANT that you use @daf.data_function!
################################################################################
↪########


@daf.data_function
def get_data(parameter):
    l_time = datetime.datetime.now()
    return f"Parameter: {parameter}\nTimestamp: {l_time.day}.{l_time.month}.{l_time.
↪year} :: {l_time.hour}:{l_time.minute}:{l_time.second}"

accounts = [
    daf.ACCOUNT( # ACCOUNT 1
        "JJJKHSAJDHKJHDKJ",
```

(continues on next page)

(continued from previous page)

```
        False,
        [
            daf.GUILD(123456789,
                      [
                          daf.TextMESSAGE(None, timedelta(seconds=15), get_data(123),␣
↪[12345, 6789]),
                      ],
                      True)
        ]
    ),
    daf.ACCOUNT( # ACCOUNT 2
        token="JJJKHSAJDHKJHDKJ",
        is_user=False,
        servers=[
            daf.GUILD(
                snowflake=123456789, # ID of server (guild) or a discord.Guild␣
↪object
                messages=[         # List MESSAGE objects
                    daf.TextMESSAGE(
                                      start_period=None,                    # If None,
↪ messages will be send on a fixed period (end period)
                                      end_period=timedelta(seconds=15),       # If␣
↪start_period is None, it dictates the fixed sending period,
                                                                              # If␣
↪start period is defined, it dictates the maximum limit of randomized period
                                      data=get_data(123),                   # Data␣
↪you want to sent to the function (Can be of types : str, embed, file, list of␣
↪types to the left
                                                                              # or␣
↪function that returns any of above types(or returns None if you don't have any␣
↪data to send yet),
                                                                              # where␣
↪if you pass a function you need to use the daf.FUNCTION decorator on top of it ).
                                      channels=[12323,2313],
                                      mode="send",                          # "send"␣
↪will send a new message every time, "edit" will edit the previous message, "clear-
↪send" will delete
                                                                              # the␣
↪previous message and then send a new one
                                      start_in=timedelta(seconds=0),        # Start␣
↪sending now (True) or wait until period
                                      remove_after=None                     # Remove␣
↪the message never or after n-times, after specific date or after timedelta
                                      ),
                ],
                logging=True,       # Generate file log of sent messages (and␣
↪failed attempts) for this user
                remove_after=None   # When to remove the guild and it's message from␣
↪the shilling list
            )
        ]
    )
```

(continues on next page)

(continued from previous page)

```
]

##############################################################################################
↪########

daf.run(accounts=accounts)
```

FILE

class daf.dtypes.**FILE**(*filename: str*)

FILE object used as a data parameter to the MESSAGE objects. This is needed opposed to a normal file object because this way, you can edit the file after the framework has already been started.

> **Warning:** This is used for sending an actual file and **NOT it's contents as text**.

Parameters
filename (*str*) – Path to the file you want sent.

AUDIO

class daf.dtypes.**AUDIO**(*filename: str*)

Used for streaming audio from file or YouTube.

> **Note:** Using a youtube video, will cause the shilling start to be delayed due to youtube data extraction.

Parameters
filename (*str*) – Path to the file you want streamed or a YouTube video url.

Raises
ValueError – Raised when the file or youtube url is not found.

to_dict()

Returns dictionary representation of this data type.

Changed in version v2.0: Changed to method `to_dict` from property `filename`

4.2.3 Auto objects

AutoCHANNEL

class daf.message.**AutoCHANNEL**(*include_pattern: str*, *exclude_pattern: Optional[str] = None*, *interval: Optional[timedelta] = datetime.timedelta(seconds=300)*)

New in version v2.3.

Used for creating instances of automatically managed channels. The objects created with this will automatically add new channels at creation and dynamically while the framework is already running, if they match the patterns.

Listing 4.5: Usage

```
# TextMESSAGE is used here, but works for others too
daf.message.TextMESSAGE(
     ..., # Other parameters
     channels=daf.message.AutoCHANNEL(...)
 )
```

Parameters

- **include_pattern** (*str*) – Regex pattern to match for the channel to be considered.
- **exclude_pattern** (*str*) – Regex pattern to match for the channel to be excluded from the consideration.

Note: If both include_pattern and exclude_pattern yield a match, the guild will be excluded from match.

- **interval** (*Optional[timedelta] = timedelta(minutes=5)*) – Interval at which to scan for new channels.

property channels: List[Union[TextChannel, VoiceChannel]]

Property that returns a list of discord.TextChannel or discord.VoiceChannel (depends on the xMESSAGE type this is in) objects in cache.

async initialize(*parent*, *channel_type: str*)

Initializes async parts of the instance. This method should be called by parent.

Parameters

- **parent** (*message.BaseMESSAGE*) – The message object this AutoCHANNEL instance is in.
- **channel_type** (*str*) – The channel type to look for when searching for channels

remove(*channel: Union[TextChannel, VoiceChannel]*)

Removes channel from cache.

Parameters

channel (*Union[discord.TextChannel, discord.VoiceChannel]*) – The channel to remove from cache.

Raises

KeyError – The channel is not in cache.

async update(***kwargs*)

Updates the object with new initialization parameters.

Parameters

kwargs (*Any*) – Any number of keyword arguments that appear in the object initialization.

Raises

Any – Raised from *initialize()* method.

AutoGUILD

class daf.guild.**AutoGUILD**(*include_pattern: str*, *exclude_pattern: Optional[str] = None*, *remove_after: Optional[Union[timedelta, datetime]] = None*, *messages: Optional[List[BaseMESSAGE]] = []*, *logging: Optional[bool] = False*, *interval: Optional[timedelta] = datetime.timedelta(seconds=600)*, *auto_join: Optional[*GuildDISCOVERY*] = None*)

Changed in version v2.5: Added `auto_join` parameter.

Internally automatically creates *daf.guild.GUILD* objects. Can also automatically join new guilds (`auto_join` parameter)

Caution: Any objects passed to AutoGUILD get **deep-copied** meaning, those same objects **will not be initialized** and cannot be used to obtain/change information regarding AutoGUILD.

Listing 4.6: Illegal use of AutoGUILD

```
auto_ch = daf.AutoCHANNEL(...)
tm = daf.TextMESSAGE(..., channels=auto_ch)

await daf.add_object(AutoGUILD(..., messages=[tm]))

auto_ch.channels # Illegal results in exception
await tm.update(...) # Illegal results in exception
```

To actually modify message/channel objects inside AutoGUILD, you need to iterate thru each GUILD.

Listing 4.7: Modifying AutoGUILD messages

```
aguild = daf.AutoGUILD(..., messages=[tm])
await daf.add_object(aguild)

for guild in aguild.guilds:
    for message in guild.messages
        await message.update(...)
```

Parameters

- **include_pattern** (*str*) – Regex pattern to use for searching guild names that are to be included. This is also checked before joining a new guild if `auto_guild` is given.
- **exclude_pattern** (*Optional[str] = None*) – Regex pattern to use for searching guild names that are **NOT** to be excluded.

  Note: If both include_pattern and exclude_pattern yield a match, the guild will be excluded from match.

- **remove_after** (*Optional[Union[timedelta, datetime]] = None*) – When to remove this object from the shilling list.
- **logging** (*Optional[bool] = False*) – Set to True if you want the guilds generated to log sent messages.
- **interval** (*Optional[timedelta] = timedelta(minutes=10)*) – Interval at which to scan for new guilds

- **auto_join** (*Optional[web.GuildDISCOVERY] = None*) – New in version v2.5.

  Optional *GuildDISCOVERY* object which will automatically discover and join guilds though the browser. This will open a Google Chrome session.

property guilds: List[*GUILD*]

Returns cached found GUILD objects.

property created_at: datetime

Returns the datetime of when the object has been created.

property deleted: bool

Indicates the status of deletion.

Returns

- *True* – The object is no longer in the framework and should no longer be used.
- *False* – Object is in the framework in normal operation.

async initialize(*parent: Any*)

Initializes the object.

Raises

ValueError – Auto-join guild functionality requires the account to be provided with username and password.

async add_message(*message: BaseMESSAGE*)

Adds a copy of the passed message to each guild inside cache.

Parameters

message (*message.BaseMESSAGE*) – Message to add.

Raises

Any – Any exception raised in *daf.guild.GUILD.add_message()*.

remove_message(*message: BaseMESSAGE*)

Removes message from the messages list.

Parameters

message (*BaseMESSAGE*) – The message to remove.

Raises

ValueError – The message is not present in the list.

async update(*init_options={}*, ***kwargs*)

Updates the object with new initialization parameters.

> **Warning:** After calling this method the entire object is reset (this includes it's GUILD objects in cache).

4.2.4 Messages

TextMESSAGE

class daf.message.**TextMESSAGE**(*start_period: Optional[Union[int, timedelta]]*, *end_period: Union[int, timedelta]*, *data: Union[str, Embed, FILE, Iterable[Union[str, Embed, FILE]]*, *_FunctionBaseCLASS]*, *channels: Union[Iterable[Union[int, TextChannel, Thread]]*, AutoCHANNEL*]*, *mode: Optional[Literal['send', 'edit', 'clear-send']] = 'send'*, *start_in: Optional[Union[timedelta, bool]] = datetime.timedelta(0)*, *remove_after: Optional[Union[int, timedelta, datetime]] = None*)

This class is used for creating objects that represent messages which will be sent to Discord's TEXT CHANNELS.

Changed in version v2.3:

Slow mode period handling
When the period is lower than the remaining time, the framework will start incrementing the original period by original period until it is larger then the slow mode remaining time.

Parameters

- **start_period** (*Union[int, timedelta, None]*) – The value of this parameter can be:
  - None - Use this value for a fixed (not randomized) sending period
  - timedelta object - object describing time difference, if this is used, then the parameter represents the bottom limit of the **randomized** sending period.
- **end_period** (*Union[int, timedelta]*) – If `start_period` is not None, then this represents the upper limit of randomized time period in which messages will be sent. If `start_period` is None, then this represents the actual time period between each message send.

> **Caution:** When the period is lower than the remaining time, the framework will start incrementing the original period by original period until it is larger then the slow mode remaining time.

Listing 4.8: **Randomized** sending period between **5** seconds and **10** seconds.

```
# Time between each send is somewhere between 5 seconds and 10␣
↪seconds.
daf.TextMESSAGE(start_period=timedelta(seconds=5), end_
↪period=timedelta(seconds=10), data="Second Message",
                channels=[12345], mode="send", start_
↪in=timedelta(seconds=0))
```

Listing 4.9: **Fixed** sending period at **10** seconds

```
# Time between each send is exactly 10 seconds.
daf.TextMESSAGE(start_period=None, end_period=timedelta(seconds=10),␣
↪data="Second Message",
                channels=[12345], mode="send", start_
↪in=timedelta(seconds=0))
```

- **data** (*Union[str, discord.Embed, FILE, List[Union[str, discord.Embed, FILE]], _FunctionBaseCLASS]*) – The data parameter is the actual data that will be sent using discord's API. The data types of this parameter can be:
  - str (normal text),
  - discord.Embed,
  - *FILE*,
  - List/Tuple containing any of the above arguments (There can up to 1 string, up to 1 discord.Embed and up to 10 *FILE* objects.
  - Function that accepts any amount of parameters and returns any of the above types. To pass a function, YOU MUST USE THE *data_function* decorator on the function before passing the function to the daf.
- **channels** (*Union[Iterable[Union[int, discord.TextChannel, discord.Thread]], daf.message.AutoCHANNEL]*) – Changed in version v2.3: Can also be *AutoCHANNEL*

  Channels that it will be advertised into (Can be snowflake ID or channel objects from PyCord).
- **mode** (*Optional[str]*) – Parameter that defines how message will be sent to a channel. It can be:
  - "send" - each period a new message will be sent,
  - "edit" - each period the previously send message will be edited (if it exists)
  - "clear-send" - previous message will be deleted and a new one sent.
- **start_in** (*Optional[timedelta]*) – When should the message be first sent.
- **remove_after** (*Optional[Union[int, timedelta, datetime]]*) – Deletes the message after:
  - int - provided amounts of sends
  - timedelta - the specified time difference
  - datetime - specific date & time

generate_log_context(*text: Optional[str]*, *embed: Embed*, *files: List[*FILE*]*, *succeeded_ch: List[Union[TextChannel, Thread]]*, *failed_ch: List[Dict[str, Any]]*) → Dict[str, Any]

Generates information about the message send attempt that is to be saved into a log.

Parameters

- **text** (*str*) – The text that was sent.
- **embed** (*discord.Embed*) – The embed that was sent.
- **files** (*List[FILE]*) – List of files that were sent.
- **succeeded_ch** (*List[Union[discord.TextChannel, discord.Thread]]*) – List of the successfully streamed channels.
- **failed_ch** (*failed_ch: List[Dict[Union[discord.TextChannel, discord.Thread], Exception]]*) – List of dictionaries contained the failed channel and the Exception object.

Returns

```
{
    sent_data:
    {
        text: str - The text that was sent,
        embed: Dict[str, Any] - The embed that was sent,
        files: List[str] - List of files that were sent
    },

    channels:
    {
        successful:
        {
            id: int - Snowflake id,
            name: str - Channel name
        },
        failed:
        {
            id: int - Snowflake id,
            name: str - Channel name,
            reason: str - Exception that caused the error
        }
    },
    type: str - The type of the message, this is always TextMESSAGE,
    mode: str - The mode used to send the message (send, edit, clear-
↪send)
}
```

Return type
Dict[str, Any]

async initialize(*parent: Any*)

This method initializes the implementation specific API objects and checks for the correct channel input context.

Parameters
parent (daf.guild.GUILD) – The GUILD this message is in

Raises

- **TypeError** – Channel contains invalid channels
- **ValueError** – Channel does not belong to the guild this message is in.
- **ValueError** – No valid channels were passed to object"

async update(*_init_options: Optional[dict] = {}*, ***kwargs: Any*)

New in version v2.0.

Used for changing the initialization parameters the object was initialized with.

Warning: Upon updating, the internal state of objects get's reset, meaning you basically have a brand new created object.

Parameters
****kwargs** (*Any*) – Custom number of keyword parameters which you want to update, these can be anything that is available during the object creation.

Raises

- **TypeError** – Invalid keyword argument was passed
- **Other** – Raised from .initialize() method.

property created_at: datetime

Returns the datetime of when the object was created

property deleted: bool

Indicates the status of deletion.

Returns

- *True* – The object is no longer in the framework and should no longer be used.
- *False* – Object is in the framework in normal operation.

DirectMESSAGE

class daf.message.**DirectMESSAGE**(*start_period: Optional[Union[int, timedelta]]*, *end_period: Union[int, timedelta]*, *data: Union[str, Embed,* FILE*, Iterable[Union[str, Embed,* FILE*]], _FunctionBaseCLASS]*, *mode: Optional[Literal['send', 'edit', 'clear-send']] = 'send'*, *start_in: Optional[Union[timedelta, bool]] = datetime.timedelta(0)*, *remove_after: Optional[Union[int, timedelta, datetime]] = None*)

This class is used for creating objects that represent messages which will be sent to Discord's TEXT CHANNELS.

Deprecated since version v2.1:

- start_in (start_now) - Using bool value to dictate whether the message should be sent at framework start.
- start_period, end_period - Using int values, use `timedelta` object instead.

Changed in version v2.1:

- start_period, end_period Accept timedelta objects.
- start_now - renamed into `start_in` which describes when the message should be first sent.
- removed `deleted` property

Parameters

- **start_period** (*Union[int, timedelta, None]*) – The value of this parameter can be:
  - None - Use this value for a fixed (not randomized) sending period
  - timedelta object - object describing time difference, if this is used, then the parameter represents the bottom limit of the **randomized** sending period.
- **end_period** (*Union[int, timedelta]*) – If `start_period` is not None, then this represents the upper limit of randomized time period in which messages will be sent. If `start_period` is None, then this represents the actual time period between each message send.

Listing 4.10: **Randomized** sending period between **5** seconds and **10** seconds.

```
# Time between each send is somewhere between 5 seconds and 10␣
↪seconds.
daf.DirectMESSAGE(
    start_period=timedelta(seconds=5), end_
↪period=timedelta(seconds=10), data="Second Message",
    mode="send", start_in=timedelta(seconds=0)
)
```

Listing 4.11: **Fixed** sending period at **10** seconds

```
# Time between each send is exactly 10 seconds.
daf.DirectMESSAGE(
    start_period=None, end_period=timedelta(seconds=10), data=
↪"Second Message",
    mode="send", start_in=timedelta(seconds=0)
)
```

- **data** (*Union[str, discord.Embed FILE, List[Union[str, discord.Embed, FILE]], _FunctionBaseCLASS]*) – The data parameter is the actual data that will be sent using discord's API. The data types of this parameter can be:
  - str (normal text),
  - discord.Embed,
  - *FILE*,
  - List/Tuple containing any of the above arguments (There can up to 1 string, up to 1 discord.Embed and up to 10 *FILE* objects.
  - Function that accepts any amount of parameters and returns any of the above types. To pass a function, YOU MUST USE THE *data_function* decorator on the function.
- **mode** (*Optional[str]*) – Parameter that defines how message will be sent to a channel. It can be:
  - "send" - each period a new message will be sent,
  - "edit" - each period the previously send message will be edited (if it exists)
  - "clear-send" - previous message will be deleted and a new one sent.
- **start_in** (*Optional[timedelta]*) – When should the message be first sent.
- **remove_after** (*Optional[Union[int, timedelta, datetime]]*) – Deletes the guild after:
  - int - provided amounts of sends
  - timedelta - the specified time difference
  - datetime - specific date & time

generate_log_context(*success_context: Dict[str, Optional[Union[bool, Exception]]], text: Optional[str], embed: Optional[Embed], files: List[*FILE*]*) → Dict[str, Any]

Generates information about the message send attempt that is to be saved into a log.

Parameters

- **text** (*str*) – The text that was sent.
- **embed** (*discord.Embed*) – The embed that was sent.
- **files** (*List[FILE]*) – List of files that were sent.
- **success_context** (*Dict[bool, Exception]*) – Dictionary containing information about succession of the DM attempt. Contains "success": *bool* key and "reason": *Exception* key which is only present if "success" is *False*

Returns

```
{
    sent_data:
    {
        text: str - The text that was sent,
        embed: Dict[str, Any] - The embed that was sent,
        files: List[str] - List of files that were sent.
    },
    success_info:
    {
        success: bool - Was sending successful or not,
        reason:  str  - If it was unsuccessful, what was the reason
    },
    type: str - The type of the message, this is always DirectMESSAGE,
    mode: str - The mode used to send the message (send, edit, clear-
↪send)
}
```

Return type
Dict[str, Any]

async initialize(*parent: Any*)

The method creates a direct message channel and returns True on success or False on failure

Changed in version v2.1: Renamed user to and changed the type from discord.User to daf.guild.USER

Parameters
parent (daf.guild.USER) – The USER this message is in

Raises
ValueError – Raised when the direct message channel could not be created

property created_at: datetime

Returns the datetime of when the object was created

property deleted: bool

Indicates the status of deletion.

Returns

- *True* – The object is no longer in the framework and should no longer be used.
- *False* – Object is in the framework in normal operation.

async update(*_init_options: Optional[dict] = {}, **kwargs*)

New in version v2.0.

Used for changing the initialization parameters the object was initialized with.

Warning: Upon updating, the internal state of objects get's reset, meaning you basically have a brand new created object.

Parameters

kwargs (*Any*) – Custom number of keyword parameters which you want to update, these can be anything that is available during the object creation.

Raises

- **TypeError** – Invalid keyword argument was passed
- **Other** – Raised from .initialize() method

VoiceMESSAGE

class daf.message.**VoiceMESSAGE**(*start_period: Optional[Union[int, timedelta]]*, *end_period: Union[int, timedelta]*, *data: Union[*AUDIO, *Iterable[*AUDIO*]*, *_FunctionBaseCLASS]*, *channels: Union[Iterable[Union[int, VoiceChannel]]*, AutoCHANNEL*]*, *volume: Optional[int] = 50*, *start_in: Optional[Union[timedelta, bool]] = datetime.timedelta(0)*, *remove_after: Optional[Union[int, timedelta, datetime]] = None*)

This class is used for creating objects that represent messages which will be streamed to voice channels.

Deprecated since version v2.1:

- start_in (start_now) - Using bool value to dictate whether the message should be sent at framework start.
- start_period, end_period - Using int values, use `timedelta` object instead.

Changed in version v2.1:

- start_period, end_period Accept timedelta objects.
- start_now - renamed into `start_in` which describes when the message should be first sent.
- removed `deleted` property

Parameters

- **start_period** (*Union[int, timedelta, None]*) – The value of this parameter can be:
  - None - Use this value for a fixed (not randomized) sending period
  - timedelta object - object describing time difference, if this is used, then the parameter represents the bottom limit of the **randomized** sending period.
- **end_period** (*Union[int, timedelta]*) – If `start_period` is not None, then this represents the upper limit of randomized time period in which messages will be sent. If `start_period` is None, then this represents the actual time period between each message send.

Listing 4.12: **Randomized** sending period between **5** seconds and **10** seconds.

```
# Time between each send is somewhere between 5 seconds and 10␣
↪seconds.
daf.VoiceMESSAGE(
    start_period=timedelta(seconds=5), end_
```

(continues on next page)

(continued from previous page)

```
↪period=timedelta(seconds=10), data=daf.AUDIO("msg.mp3"),
    channels=[12345], start_in=timedelta(seconds=0), volume=50
)
```

Listing 4.13: **Fixed** sending period at **10** seconds

```
# Time between each send is exactly 10 seconds.
daf.VoiceMESSAGE(
    start_period=None, end_period=timedelta(seconds=10), data=daf.
↪AUDIO("msg.mp3"),
    channels=[12345], start_in=timedelta(seconds=0), volume=50
)
```

- **data** (AUDIO) – The data parameter is the actual data that will be sent using discord's API. The data types of this parameter can be:
  - AUDIO object.
  - Function that accepts any amount of parameters and returns an AUDIO object. To pass a function, YOU MUST USE THE *data_function* decorator on the function.
- **channels** (*Union[Iterable[Union[int, discord.VoiceChannel]], daf.message.AutoCHANNEL]*) – Changed in version v2.3: Can also be *AutoCHANNEL*

  Channels that it will be advertised into (Can be snowflake ID or channel objects from PyCord).
- **volume** (*Optional[int]*) – The volume (0-100%) at which to play the audio. Defaults to 50%. This was added in v2.0.0
- **start_in** (*Optional[timedelta]*) – When should the message be first sent.
- **remove_after** (*Optional[Union[int, timedelta, datetime]]*) – Deletes the message after:
  - int - provided amounts of sends
  - timedelta - the specified time difference
  - datetime - specific date & time

generate_log_context(*audio:* AUDIO, *succeeded_ch: List[VoiceChannel]*, *failed_ch: List[Dict[str, Any]]*) → Dict[str, Any]

Generates information about the message send attempt that is to be saved into a log.

Parameters

- **audio** (*audio*) – The audio that was streamed.
- **succeeded_ch** (*List[Union[discord.VoiceChannel]]*) – List of the successfully streamed channels
- **failed_ch** (*List[Dict[discord.VoiceChannel, Exception]]*) – List of dictionaries contained the failed channel and the Exception object

Returns

```
{
    sent_data:
```

(continues on next page)

(continued from previous page)

```
    {
        streamed_audio: str - The filename that was streamed/
↪youtube url
    },
    channels:
    {
        successful:
        {
            id: int - Snowflake id,
            name: str - Channel name
        },
        failed:
        {
            id: int - Snowflake id,
            name: str - Channel name,
            reason: str - Exception that caused the error
        }
    },
    type: str - The type of the message, this is always␣
↪VoiceMESSAGE.
}
```

Return type
Dict[str, Any]

async initialize(*parent: Any*)

This method initializes the implementation specific API objects and checks for the correct channel input context.

Parameters
parent (daf.guild.GUILD) – The GUILD this message is in

Raises

- **TypeError** – Channel contains invalid channels
- **ValueError** – Channel does not belong to the guild this message is in.
- **ValueError** – No valid channels were passed to object"

property created_at: datetime

Returns the datetime of when the object was created

property deleted: bool

Indicates the status of deletion.

Returns

- *True* – The object is no longer in the framework and should no longer be used.
- *False* – Object is in the framework in normal operation.

async update(*_init_options: Optional[dict] = {}*, ***kwargs*)

New in version v2.0.

Used for changing the initialization parameters the object was initialized with.

Warning: Upon updating, the internal state of objects get's reset, meaning you basically have a brand new created object.

Parameters
****kwargs** (*Any*) – Custom number of keyword parameters which you want to update, these can be anything that is available during the object creation.

Raises
- **TypeError** – Invalid keyword argument was passed
- **Other** – Raised from .initialize() method

4.2.5 Clients

get_accounts

daf.core.**get_accounts**() → List[*ACCOUNT*]

New in version v2.4.

Returns
List of running accounts.

Return type
List[*client.ACCOUNT*]

SeleniumCLIENT

class daf.web.**SeleniumCLIENT**(*username: str*, *password: str*, *proxy: str*)

New in version v2.5.

Client used to control the Discord web client for things such as logging in, joining guilds, passing "Complete" for guild rules.

This is created in the ACCOUNT object in case `web` parameter inside ACCOUNT is True.

Note: This is automatically created in *ACCOUNT* and is also bound to the *ACCOUNT* instance.

To retrieve it from *ACCOUNT*, use *selenium*.

Parameters
- **username** (*str*) – The Discord username to login with.
- **password** (*str*) – The Discord password to login with.
- **proxy** (*str*) – The proxy url to use when connecting to Chrome.

property token: *str*

Returns accounts's token

update_token_file() → str

Updates the tokens JSON file.

Raises
OSError – There was an error saving/reading the file.

Returns
The token.

Return type
str

async random_sleep(*bottom: int*, *upper: int*)

Sleeps randomly to prevent detection.

async async_execute(*method: Callable*, **args*)

Runs method in executor to force async.

Parameters

- **method** (*Callable*) – Callable to execute in async thread executor.
- **args** – Variadic arguments passed to method.

async random_server_click()

Randomly clicks on the servers panel to avoid CAPTCHA triggering.

async fetch_invite_link(*url: str*)

Fetches the invite link in case it is valid.

Parameters
url (*str | None*) – The url to check or None if error ocurred/invalid link.

async slow_type(*form: WebElement*, *text: str*)

Slowly types into a form to prevent detection.

Parameters

- **form** (*WebElement*) – The form to type text into.
- **text** (*str*) – The text to type in the form.

async slow_clear(*form: WebElement*)

Slowly deletes the text from an input

Parameters
form (*WebElement*) – The form to delete text from.

async await_url_change()

Waits for url to change.

Raises
TimeoutError – Waited for too long.

async await_load()

Waits for the Discord spinning logo to disappear, which means that the content has finished loading.

Raises
TimeoutError – The page loading timed-out.

async await_captcha()

Waits for CAPTCHA to be completed.

Raises
TimeoutError – CAPTCHA was not solved in time.

async initialize() → None

Starts the webdriver whenever the framework is started.

Raises
Any – Raised in *login()* method.

async login() → str

Logins to Discord using the webdriver and saves the account token to JSON file.

Returns
Token belonging to provided username.

Return type
str

Raises

- TimeoutError – Raised when any of the await_* methods timed-out.
- RuntimeError – Unable to login due to internal exception.

async hover_click(*element: WebElement*)

Hovers an element and clicks on it.

Parameters
element (*WebElement*) – The element to hover click.

async join_guild(*invite: str*) → None

Joins the guild thru the browser.

Parameters
invite (*str*) – The invite link/code of the guild to join.

Raises

- RuntimeError – Internal error ocurred.
- RuntimeError – The user is banned from the guild.
- TimeoutError – Timed out while waiting for actions to complete.

ACCOUNT

class daf.client.ACCOUNT(*token: Optional[str] = None*, *is_user: Optional[bool] = False*, *intents: Optional[Intents] = None*, *proxy: Optional[str] = None*, *servers: Optional[List[Union[GUILD, USER, AutoGUILD]]] = None*, *username: Optional[str] = None*, *password: Optional[str] = None*)

New in version v2.4.

Changed in version v2.5: Added username and password parameters. For logging in automatically

Represents an individual Discord account.

Each ACCOUNT instance runs it's own shilling task.

Parameters

- **token** (*str*) – The Discord account's token
- **is_user** (*Optional[bool] =False*) – Declares that the `token` is a user account token ("self-bot")
- **intents** (*Optional[discord.Intents]=discord.Intents.default()*) – Discord Intents (settings of events that the client will subscribe to)
- **proxy** (*Optional[str]=None*) – The proxy to use when connecting to Discord.

Important: It is **RECOMMENDED** to use a proxy if you are running **MULTIPLE** accounts. Running multiple accounts from the same IP address, can result in Discord detecting self-bots.

Running multiple bot accounts on the other hand is perfectly fine without a proxy.

- **servers** (*Optional[List[guild.GUILD | guild.USER | guild.AutoGUILD]]=[]*) – Predefined list of servers (guilds, users, auto-guilds).
- **username** (*Optional[str]*) – The username to login with.
- **password** (*Optional[str]*) – The password to login with.

Raises

- **ModuleNotFoundError** – 'proxy' parameter was provided but requirements are not installed.
- **ValueError** – 'token' is not allowed if 'username' is provided and vice versa.

property selenium: *SeleniumCLIENT*

New in version v2.5.

Returns the, bound to account, Selenium client

property running: bool

Is the account still running?

Returns

- *True* – The account is logged in and shilling is active.
- *False* – The shilling has ended or not begun.

property deleted: bool

Indicates the status of deletion.

Returns

- *True* – The object is no longer in the framework and should no longer be used.
- *False* – Object is in the framework in normal operation.

property servers

Returns all guild like objects inside the account's s shilling list. This also includes *AutoGUILD*

property client: Client

Returns the API wrapper client

async initialize()

Initializes the API wrapper client layer.

Raises
RuntimeError – Unable to login to Discord.

async add_server(*server: Union[GUILD, USER, AutoGUILD]*)

Initializes a guild like object and adds it to the internal account shill list.

Parameters
server (guild.GUILD | guild.USER | guild.AutoGUILD) – The guild like object to add

Raises
Any – Raised in *daf.guild.GUILD.initialize()* | *daf.guild.USER.initialize()* | *daf.guild.AutoGUILD.initialize()*

remove_server(*server: Union[GUILD, USER, AutoGUILD]*)

Removes a guild like object from the shilling list.

Parameters
server (guild.GUILD | guild.USER | guild.AutoGUILD) – The guild like object to remove

Raises
ValueError – server is not in the shilling list.

get_server(*snowflake: Union[int, Guild, User, Object]*) → Optional[Union[*GUILD*, *USER*]]

Retrieves the server based on the snowflake id or discord API object.

Parameters
snowflake (*Union[int, discord.Guild, discord.User, discord.Object]*) – Snowflake ID or Discord API object.

Returns
- *Union[guild.GUILD, guild.USER]* – The DAF server object.
- *None* – The object was not found.

async update(***kwargs*)

Updates the object with new parameters and afterwards updates all lower layers (GUILD->MESSAGE->CHANNEL).

> **Warning:** After calling this method the entire object is reset.

4.2.6 Web

QuerySortBy

enum daf.web.QuerySortBy(*value*)

Enumerated options that can be passed to the sort_by parameter of *daf.web.GuildDISCOVERY*.

Valid values are as follows:

TEXT_RELEVANCY = <QuerySortBy.TEXT_RELEVANCY: 0>

TOP = <QuerySortBy.TOP: 1>

RECENTLY_CREATED = <QuerySortBy.RECENTLY_CREATED: 2>

TOP_VOTED = <QuerySortBy.TOP_VOTED: 3>

TOTAL_USERS = <QuerySortBy.TOTAL_USERS: 4>

QueryMembers

enum daf.web.**QueryMembers**(*value*)

Enumerated options that can be passed to the `total_members` parameter of *daf.web.GuildDISCOVERY*.

Valid values are as follows:

ALL = <QueryMembers.ALL: 0>

SUB_100 = <QueryMembers.SUB_100: (0, 100)>

B100_1k = <QueryMembers.B100_1k: (100, 1000)>

B1k_10k = <QueryMembers.B1k_10k: (1000, 10000)>

ABV_10k = <QueryMembers.ABV_10k: 1>

GuildDISCOVERY

class daf.web.**GuildDISCOVERY**(*prompt: str*, *sort_by: Optional[QuerySortBy] = QuerySortBy.TOP*, *total_members: Optional[QueryMembers] = QueryMembers.ALL*, *limit: Optional[int] = 15*)

Client used for searching servers. To be used with *daf.guild.AutoGUILD*.

Parameters

- **prompt** (*str*) – Query parameter for server search.
- **sort_by** (*Optional[QuerySortBy]*) – Query parameter for sorting method for results.
- **total_members** (*Optional[QueryMembers]*) – Query parameter for member limit.
- **limit** (*Optional[int]*) – The maximum amount of servers to query. Defaults to 15 servers.

4.2.7 Guilds

GUILD

class daf.guild.**GUILD**(*snowflake: Union[int, Guild]*, *messages: Optional[List[Union[TextMESSAGE, VoiceMESSAGE]]] = []*, *logging: Optional[bool] = False*, *remove_after: Optional[Union[timedelta, datetime]] = None*)

The GUILD object represents a server to which messages will be sent.

Changed in version v2.1:

- Added `created_at` attribute
- Added `remove_after` parameter

Parameters

- **snowflake** (*Union[int, discord.Guild]*) – Discord's snowflake ID of the guild or discord.Guild object.
- **messages** (*Optional[List[Union[TextMESSAGE, VoiceMESSAGE]]]*) – Optional list of TextMESSAGE/VoiceMESSAGE objects.
- **logging** (*Optional[bool]*) – Optional variable dictating whatever to log sent messages inside this guild.
- **remove_after** (*Optional[Union[timedelta, datetime]]*) – Deletes the guild after:
  - timedelta - the specified time difference
  - datetime - specific date & time

async initialize(*parent: Any*) → None

This function initializes the API related objects and then tries to initialize the MESSAGE objects.

Note: This should NOT be manually called, it is called automatically after adding the message.

Raises

- **ValueError** – Raised when the guild_id wasn't found.
- **Other** – Raised from .add_message(message_object) method.

async update(*init_options={}*, ***kwargs*)

Used for changing the initialization parameters, the object was initialized with.

New in version v2.0.

Warning: Upon updating, the internal state of objects get's reset, meaning you basically have a brand new created object. It also resets the message objects.

Parameters

****kwargs** (*Any*) – Custom number of keyword parameters which you want to update, these can be anything that is available during the object creation.

Raises

- **TypeError** – Invalid keyword argument was passed.
- **Other** – Raised from .initialize() method.

async add_message(*message: BaseMESSAGE*)

Adds a message to the message list.

Warning: To use this method, the guild must already be added to the framework's shilling list (or initialized).

Parameters

message (*BaseMESSAGE*) – Message object to add.

Raises

- **TypeError** – Raised when the message is not of type the guild allows.
- **Other** – Raised from message.initialize() method.

property apiobject: Object

New in version v2.4.

Returns the Discord API wrapper's object of self.

property created_at: datetime

New in version v2.1.

Returns the datetime of when the object has been created.

property deleted: bool

Indicates the status of deletion.

Returns

- *True* – The object is no longer in the framework and should no longer be used.
- *False* – Object is in the framework in normal operation.

generate_log_context() → Dict[str, Union[str, int]]

Generates a dictionary of the guild's context, which is then used for logging.

Return type
Dict[str, Union[str, int]]

property messages: List[BaseMESSAGE]

Returns all the (initialized) message objects inside the object.

New in version v2.0.

remove_message(*message: BaseMESSAGE*)

Removes a message from the message list.

Parameters
message (*BaseMESSAGE*) – Message object to remove.

Raises

- **TypeError** – Raised when the message is not of type the guild allows.
- **ValueError** – Raised when the message is not present in the list.

property snowflake: int

New in version v2.0.

Returns the discord's snowflake ID.

USER

class daf.guild.**USER**(*snowflake: Union[int, User]*, *messages: Optional[List[*DirectMESSAGE*]] = []*, *logging: Optional[bool] = False*, *remove_after: Optional[Union[timedelta, datetime]] = None*)

The USER object represents a user to whom messages will be sent.

Changed in version v2.1:

- Added `created_at` attribute
- Added `remove_after` parameter

Parameters

- **snowflake** (*Union[int, discord.User]*) – Discord's snowflake ID of the user or discord.User object.
- **messages** (*Optional[List[DirectMESSAGE]]*) – Optional list of DirectMESSAGE objects.
- **logging** (*Optional[bool]*) – Optional variable dictating whatever to log sent messages inside this guild.
- **remove_after** (*Optional[Union[timedelta, datetime]]*) – Deletes the user after:
  - timedelta - the specified time difference
  - datetime - specific date & time

async initialize(*parent: Any*)

This function initializes the API related objects and then tries to initialize the MESSAGE objects.

Raises

- **ValueError** – Raised when the DM could not be created.
- **Other** – Raised from .add_message(message_object) method.

async update(*init_options={}*, ***kwargs*)

New in version v2.0.

Used for changing the initialization parameters, the object was initialized with.

> **Warning:** Upon updating, the internal state of objects get's reset, meaning you basically have a brand new created object. It also resets the message objects.

Parameters

****kwargs** (*Any*) – Custom number of keyword parameters which you want to update, these can be anything that is available during the object creation.

Raises

- **TypeError** – Invalid keyword argument was passed.
- **Other** – Raised from .initialize() method.

async add_message(*message: BaseMESSAGE*)

Adds a message to the message list.

> **Warning:** To use this method, the guild must already be added to the framework's shilling list (or initialized).

Parameters

message (*BaseMESSAGE*) – Message object to add.

Raises

- **TypeError** – Raised when the message is not of type the guild allows.
- **Other** – Raised from message.initialize() method.

property apiobject: Object

New in version v2.4.

Returns the Discord API wrapper's object of self.

property created_at: datetime

New in version v2.1.

Returns the datetime of when the object has been created.

property deleted: bool

Indicates the status of deletion.

Returns

- *True* – The object is no longer in the framework and should no longer be used.
- *False* – Object is in the framework in normal operation.

generate_log_context() → Dict[str, Union[str, int]]

Generates a dictionary of the guild's context, which is then used for logging.

Return type
Dict[str, Union[str, int]]

property messages: List[BaseMESSAGE]

Returns all the (initialized) message objects inside the object.

New in version v2.0.

remove_message(*message: BaseMESSAGE*)

Removes a message from the message list.

Parameters
message (*BaseMESSAGE*) – Message object to remove.

Raises

- **TypeError** – Raised when the message is not of type the guild allows.
- **ValueError** – Raised when the message is not present in the list.

property snowflake: int

New in version v2.0.

Returns the discord's snowflake ID.

4.2.8 DAF control reference

initialize

async daf.core.**initialize**(*token: Optional[str] = None*, *server_list: Optional[List[Union[GUILD, USER, AutoGUILD]]] = None*, *is_user: Optional[bool] = False*, *user_callback: Optional[Union[Callable, Coroutine]] = None*, *server_log_output: Optional[str] = None*, *sql_manager: Optional[LoggerSQL] = None*, *intents: Optional[Intents] = None*, *debug: Optional[Union[TraceLEVELS, int, str, bool]] = TraceLEVELS.NORMAL*, *proxy: Optional[str] = None*, *logger: Optional[LoggerBASE] = None*, *accounts: Optional[List[ACCOUNT]] = []*) → None

The main initialization function. It initializes all the other modules, creates advertising tasks and initializes all the core functionality. If you want to control your own event loop, use this instead of run.

Parameters

Any (*Any*) – Parameters are the same as in *daf.core.run()*.

shutdown

async daf.core.**shutdown**(*loop: Optional[AbstractEventLoop] = None*) → None

Stops the framework.

Parameters

loop (*Optional[asyncio.AbstractEventLoop]*) – The loop everything is running in. Leave empty for default loop.

run

daf.core.**run**(*token: Optional[str] = None*, *server_list: Optional[List[Union[*GUILD, USER, AutoGUILD*]]] = None*, *is_user: Optional[bool] = False*, *user_callback: Optional[Union[Callable, Coroutine]] = None*, *server_log_output: Optional[str] = None*, *sql_manager: Optional[*LoggerSQL*] = None*, *intents: Optional[Intents] = None*, *debug: Optional[Union[*TraceLEVELS*, int, str, bool]] = TraceLEVELS.NORMAL*, *proxy: Optional[str] = None*, *logger: Optional[*LoggerBASE*] = None*, *accounts: Optional[List[*ACCOUNT*]] = []*) → None

Runs the framework and does not return until the framework is stopped (*daf.core.shutdown()*). After stopping, it returns None.

Warning: This will block until the framework is stopped, if you want manual control over the asyncio event loop, eg. you want to start the framework as a task, use the *daf.core.initialize()* coroutine.

Changed in version v2.4: Added `accounts` parameter.

Deprecated since version v2.4:

The following parameters were deprecated in favor of support for multiple accounts

- token
- is_user
- server_list
- intents
- proxy

The above parameters should be passed to *ACCOUNT*.

Parameters

- **user_callback** (*Optional[Union[Callable, Coroutine]]*) – Function or async function to call after the framework has been started.

- **debug** (*Optional[TraceLEVELS | int | str] = TraceLEVELS.NORMAL*) – Changed in version v2.3: Deprecate use of bool (assume TraceLEVELS.NORMAL). Add support for TraceLEVELS or int or str that converts to TraceLEVELS.

  The level of trace for trace function to display. The higher value this option is, the more will be displayed.

- **logger** (*Optional[loggers.LoggerBASE]*) – The logging class to use. If this is not provided, JSON is automatically used.

- **accounts** (*Optional[List[client.ACCOUNT]]*) – New in version v2.4.

  List of *ACCOUNT* (Discord accounts) to use.

Raises

- **ModuleNotFoundError** – Missing modules for the wanted functionality, install with `pip install discord-advert-framework[optional-group]`.
- **ValueError** – Invalid proxy url.

4.2.9 Dynamic mod.

add_object

async daf.core.**add_object**(*obj: <class 'daf.client.ACCOUNT'>*) → None

Adds an account to the framework.

Parameters

obj (client.ACCOUNT) – The account object to add

Raises

- **ValueError** – The account has already been added to the list.
- **TypeError** – `obj` is of invalid type.

add_object

async daf.core.**add_object**(*obj: typing.Union[daf.guild.USER, daf.guild.GUILD, daf.guild.AutoGUILD],snowflake: <class 'daf.client.ACCOUNT'>*) → None

Adds a guild or an user to the daf.

Parameters

- **obj** (guild.USER | guild.GUILD | guild.AutoGUILD) – The guild object to add into the account (`snowflake`).
- **snowflake** (*client.ACCOUNT=None*) – The account to add this guild/user to.

Raises

- **ValueError** – The guild/user is already added to the daf.
- **TypeError** – The object provided is not supported for addition.
- **TypeError** – Invalid parameter type.
- **RuntimeError** – When using deprecated method of adding items to the shill list, no accounts were available.
- **Other** – Raised in the obj.initialize() method

add_object

async daf.core.**add_object**(*obj: Union[*daf.message.text_based.DirectMESSAGE, daf.message.text_based.TextMESSAGE, daf.message.voice_based.VoiceMESSAGE*]*, *snowflake: Union[*daf.guild.GUILD, daf.guild.USER*]*) → None

Deprecated since version v2.4: Using int, discord.* objects in the snowflake parameter. This functionality is planned for removal in v2.5.

Adds a message to the daf.

Parameters

- **obj** (message.DirectMESSAGE | message.TextMESSAGE | message.VoiceMESSAGE) – The message object to add into the daf.
- **snowflake** (guild.GUILD | *guild.USER a discord API wrapper object).*) – Which guild/user to add it to (can be snowflake id or a framework _BaseGUILD object or

Raises

- **TypeError** – The object provided is not supported for addition.
- **ValueError** – guild_id wasn't provided when adding a message object (to which guild should it add)
- **ValueError** – Missing snowflake parameter.
- **ValueError** – Could not find guild with that id.
- **Other** – Raised in the obj.add_message() method

remove_object

async daf.core.**remove_object**(*snowflake: Union[_BaseGUILD, BaseMESSAGE,* AutoGUILD, ACCOUNT*]*) → None

Changed in version v2.4.1: Turned async for fix bug of missing functionality

Changed in version v2.4: | Now accepts client.ACCOUNT. | Removed support for `int` and for API wrapper (PyCord) objects.

Removes an object from the daf.

Parameters

snowflake (*guild._BaseGUILD | message.BaseMESSAGE |* guild.AutoGUILD | client.ACCOUNT) – The object to remove from the framework.

Raises

- **ValueError** – Item (with specified snowflake) not in the shilling list.
- **TypeError** – Invalid argument.

4.3 Changelog

4.3.1 Info

See also:

Releases

Note: The library first started as a single file script that I didn't make versions of. When I decided to turn it into a library, I've set the version number based on the amount of commits I have made since the start.

Glossary

[Breaking change]
Means that the change will break functionality from previous version.

[Potentially breaking change]
The change could break functionality from previous versions but only if it was used in a certain way.

4.3.2 Releases

v2.5.3

- Fixed voice not working due to Discord's API changes.

v2.5.2

- Fixed `typechecked` module error

v2.5.1

- Fixed failure without SQL

v2.5

- **[Breaking change]** Removed `EMBED` object, use `daf.discord.Embed` instead.
- **[Breaking change]** Removed `timing` module since it only contained deprecated objects.
- **[Breaking change]** Minumum Python version has been bumbed to **Python v3.10**.
- WEB INTEGRATION:
  - Automatic login and (semi-automatic) guild join though *daf.web.SeleniumCLIENT*.
  - Automatic server discovery though *daf.web.GuildDISCOVERY*

v2.4.3

- Fixed missing documentation members

v2.4.2 (v2.3.4)

- Fixed channel verification bug:
  - Fixes bug where messages try to be sent into channels that have not passed verification (complete button)

v2.4

- Multiple accounts support:
  - Added *daf.client.ACCOUNT* for running multiple accounts at once. Proxies are strongly recommended!
  - Deprecated use of:
    - token,
    - is_user,
    - proxy,
    - server_list,
    - intents

    inside the *daf.core.run()* function.
  - New function *daf.core.get_accounts()* that returns the list of all running accounts in the framework.
- Deprecated *add_object()* and *remove_object()* functions accepting API wrapper objects or `int` type for the `snowflake` parameter.
- Deprecated `daf.core.get_guild_user` function due to multiple accounts support.
- Deprecated `daf.client.get_client` function due to multiple accounts support.

v2.3

- **[Breaking change]** Removed `exceptions` module, meaning that there are no DAFError derived exceptions from this version on. They are replaced with build-in Python exceptions.
- Automatic scheme generation and management:
  - *daf.guild.AutoGUILD* class for auto-managed GUILD objects.
  - *daf.message.AutoCHANNEL* class for auto-managed channels inside message.
- Debug levels:
  - Added deprecated to *TraceLEVELS*.
  - Changed the *daf.core.run()*'s debug parameter to accept a value from *TraceLEVELS*, to dictate what level trace should be displayed.
- *Messages* objects period automatically increases if it is less than slow-mode timeout.
- The *data_function*'s input function can now also be async.

v2.2

- `user_callback` parameter for function *daf.core.run()* can now also be a regular function instead of just `async`.
- Deprecated `daf.dtypes.EMBED`, use `discord.Embed` instead.
- **[Breaking change]** Removed `get_sql_manager` function.
- ***daf.core.run()*:**
  - Added `logging` parameter
  - Deprecated parameters `server_log_output` and `sql_manager`.
- Logging manager objects: LoggerJSON, LoggerCSV, LoggerSQL
- New *daf.logging.get_logger()* function for retrieving the logger object used.
- *daf.core.initialize()* for manual control of asyncio (same as *daf.core.run()* except it is async)
- **SQL:**
  - SQL logging now supports **Microsoft SQL Server, MySQL, PostgreSQL and SQLite databases**.
  - **[Breaking change]** *LoggerSQL*'s parameters are re-arranged, new parameters of which, the `dialect` (mssql, sqlite, mysql, postgresql) parameter must be passed.
- **Development:**
  - `doc_category` decorator for automatic documentation
  - Removed `common` module and moved constants to appropriate modules

v2.1.4

Bug fixes:

- `Fix incorrect parameter name in documentation`.

v2.1.3

Bug fixes:

- `[Bug]: KeyError: 'code' on rate limit #198`.

v2.1.2

Bug fixes:

- #195 VoiceMESSAGE did not delete deleted channels.
- Exception on initialization of static server list in case any of the messages had failed their initialization.

v2.1.1

- Fixed `[Bug]: Predefined servers' errors are not suppressed #189`.
- Support for readthedocs.

v2.1

- Changed the import `import framework` to `import daf`. Using `import framework` is now deprecated.
- **`remove_after` parameter:**
  Classes: *daf.guild.GUILD*, *daf.guild.USER*, *daf.message.TextMESSAGE*, *daf.message.VoiceMESSAGE*, *daf.message.DirectMESSAGE*

  now support the remove_after parameter which will remove the object from the shilling list when conditions met.
- **Proxies:**
  Added support for using proxies. To use a proxy pass the `daf.run()` function with a `proxy` parameter
- **discord.EmbedField:**
  [Breaking change] Replaced discord.EmbedField with discord.EmbedField.
- **timedelta:**
  start_period and end_period now support `timedelta` object to specify the send period. Use of `int` is deprecated

  [Potentially breaking change] Replaced `start_now` with `start_in` parameter, deprecated use of bool value.
- **Channel checking:**
  `daf.TextMESSAGE` and `daf.VoiceMESSAGE` now check if the given channels are actually inside the guild
- **Optionals:**
  [Potentially breaking change] Made some functionality optional: `voice`, `proxy` and `sql` - to install use `pip install discord-advert-framework[dependency here]`
- **CLIENT:**
  [Breaking change] Removed the CLIENT object, discord.Client is now used as the CLIENT class is no longer needed due to improved startup
- **Bug fixes:**

  Time slippage correction:
  This occurred if too many messages were ready at once, which resulted in discord's rate limit, causing a permanent slip.

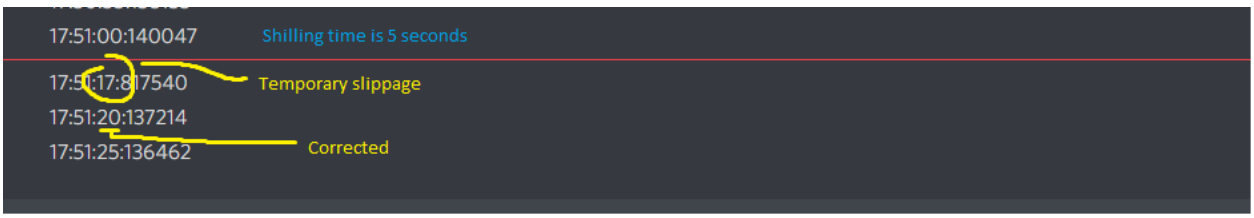


Fig. 4.3: Time slippage correction

  Slow mode correction:
  Whenever a channel was in slow mode, it was not properly handled. This is now fixed.

v2.0

- New cool looking web documentation (the one you're reading now)
- Added volume parameter to `daf.VoiceMESSAGE`
- Changed `channel_ids` to `channels` for `daf.VoiceMESSAGE` and `daf.TextMESSAGE`. It can now also accept discord.<Type>Channel objects.
- Changed `user_id`/ `guild_id` to `snowflake` in `daf.GUILD` and `daf.USER`. This parameter now also accept discord.Guild (`daf.GUILD`) and discord.User (`daf.USER`)
- Added `.update` method to some objects for allowing dynamic modifications of initialization parameters.
- `daf.AUDIO` now also accepts a YouTube link for streaming YouTube videos.
- New Exceptions system - most functions now raise exceptions instead of just returning bool to allow better detection of errors.
- Bug fixes and other small improvements.

v1.9.0

- Added support for logging into a SQL database (MS SQL Server only). See *Relational Database Log (SQL)*.
- `daf.run()` function now accepts discord.Intents.
- `daf.add_object()` and `daf.remove_object()` functions created to allow for dynamic modification of the shilling list.
- Other small improvements.

v1.8.1

- JSON file logging.
- Automatic channel removal if channel get's deleted and message removal if all channels are removed.
- Improved debug messages.

v1.7.9

- `daf.DirectMESSAGE` and `daf.USER` classes created for direct messaging.

INDEX

I

J

L

M

N

P

R

S

T

U

V

W