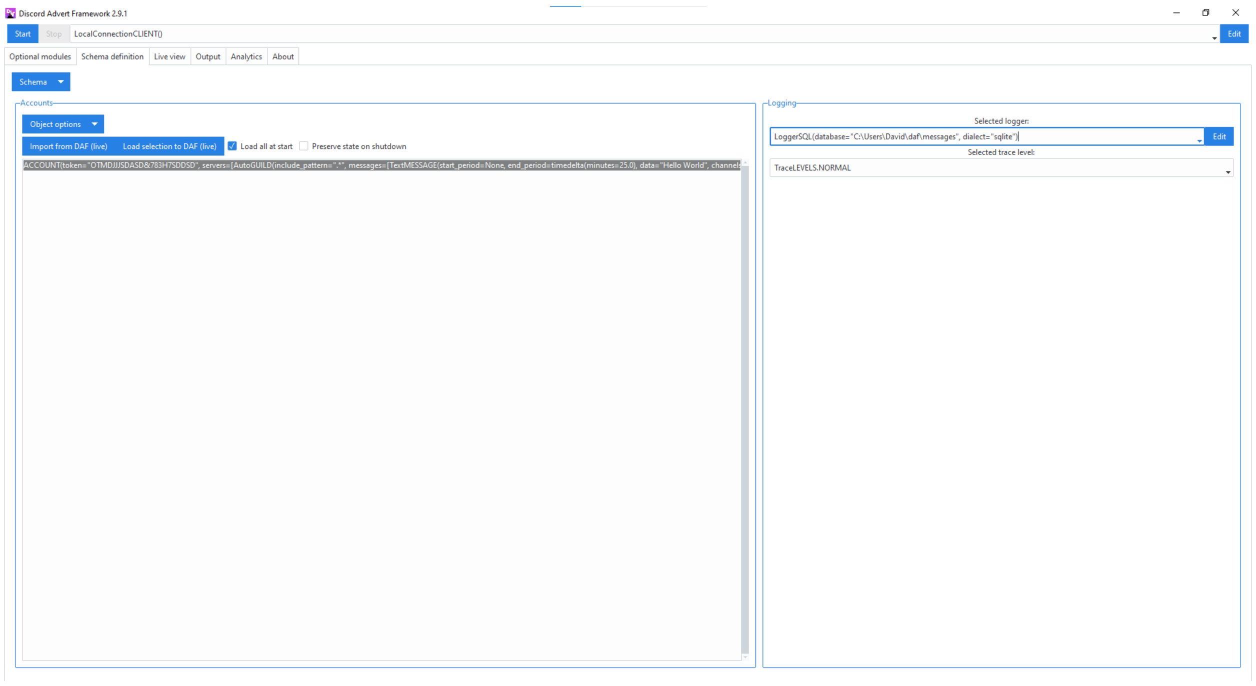
Discord Advertisement Framework

David Hozic

Mar 02, 2024

CONTENTS

The Discord advertisement framework is a Python based automatic application that allows **easy automatic advertisement** (and much more) on Discord.



LocalConnectionCLIENT()

✓ Load all at start

Preserve state on shutdown

```
ACCOUNT(token="OTMDJJJSDASD&783H7SDDSD", servers=[AutoGUILD(include_pattern=".*", messages=[TextMESSAGE(start_period=None, end_period=timedelta(minutes=25.0), data="Hello World", channels
```

Selected logger:

```
LoggerSQL(database="C:\Users\David\daf\messages", dialect="sqlite")
```

Selected trace level:

```
TraceLEVELS.NORMAL
```

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

NEED HELP?

- Checkout the guides:
  - *Guide (GUI)*
  - *Guide (core)*
- Contact me in my Discord server.

CHAPTER THREE

KEY FEATURES

- Automatic periodic and scheduled messages to multiple servers and channels,
- Error checking and recovery,
- Message logging, invite link tracking & statistics
- Multi-account support
- Graphical Interface (GUI) / Console (script)
- Easy to setup
- Programmatic usage
- Much more

Note: Running on user accounts is against Discord ToS, however DAF still enables it.

CHAPTER

FOUR

INSTALLATION

DAF can be installed in one of the two ways:

1. As a standalone EXE on Windows. It can be downloaded in releases. Please note that antivirus software might detect this as a virus. This is because the exe bundles a Python interpreter alongside the Python package. If that happens, disable your antivirus or consider installing the software as a Python package (explained in the following bullet point).
2. Installed as a Python package through the terminal / PowerShell / cmd.

Main package

Pre-requirement: Python (minimum v3.9; recommended 3.11)

```
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 4.1: Voice Messaging / AUDIO

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

SQL

Listing 4.2: SQL logging

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

Chrome integration

Listing 4.3: Chrome integration

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

All

Install all of the (left) optional dependencies

Listing 4.4: All

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

CHAPTER

FIVE

TABLE OF CONTENTS

5.1 Guide

The following pages contain guides to running the Discord Advertisement Framework in 2 main modes:

1. *Core mode (console / programming)*
2. *Graphical mode (GUI)*

5.1.1 Guide (GUI)

This section contains the guide to using the Discord Advertisement Framework inside a **G**raphical **U**ser **I**nterface.

Quickstart (GUI)

This page contains information to quickly getting started with the GUI.

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

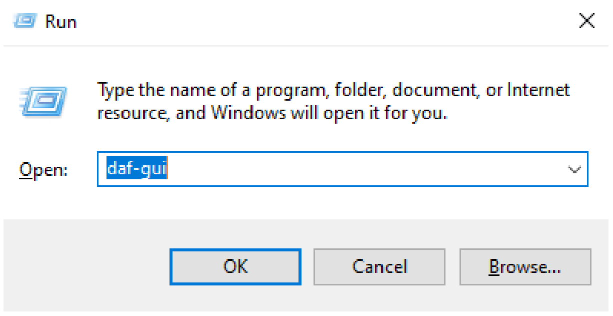
After successful installation, DAF can be run in graphical mode by executing the command `daf-gui` command inside the terminal

```
$ daf-gui
```

or by using `python -m daf_gui` command.

```
$ python -m daf_gui
```

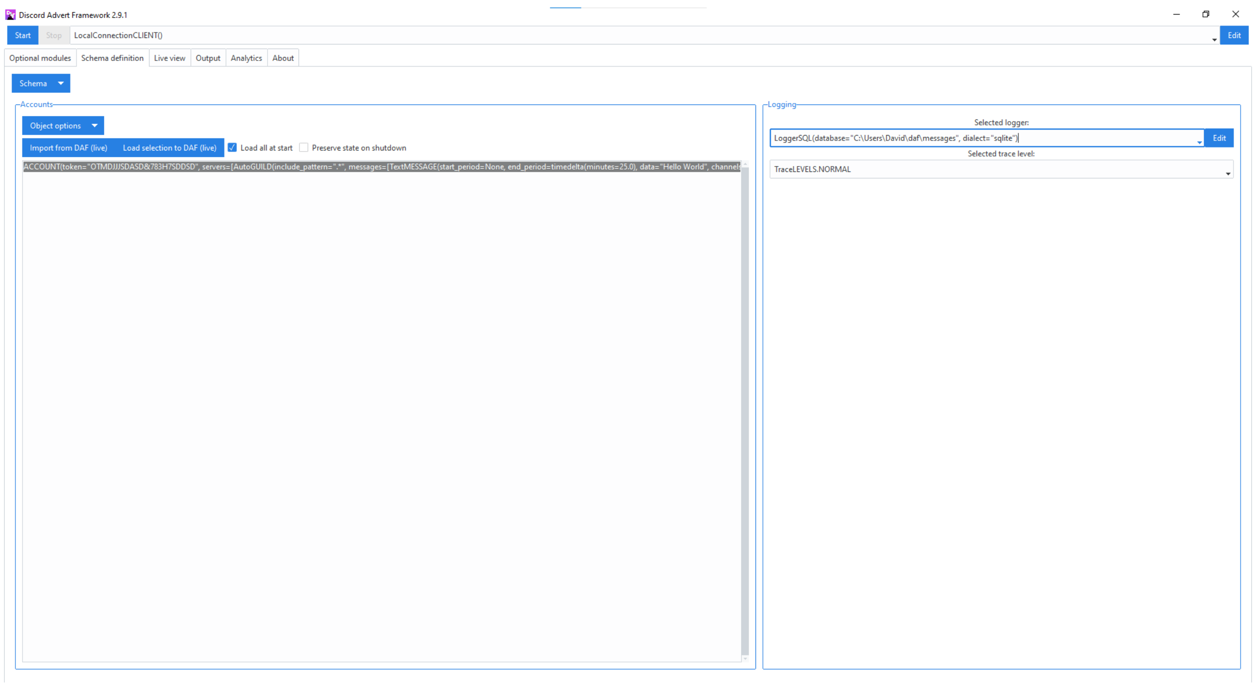
On Windows it can also be started though the Run (Win + R) menu.


Run

Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: daf-gui

This will open up (after a few seconds) a graphical display you can use to control the framework.



LocalConnectionCLIENT()

Load all at start [✓] / Preserve state on shutdown [ ]

```
ACCOUNT(token="OTMDJJJSDASD&783H7SDDSD", servers=[AutoGUILD(include_pattern=".*", messages=[TextMESSAGE(start_period=None, end_period=timedelta(minutes=25.0), data="Hello World", channels
```

Selected logger: LoggerSQL(database="C:\Users\David\daf\messages", dialect="sqlite")

Selected trace level: TraceLEVELS.NORMAL

Before going forward, please note that the GUI is just an extra layer above the actual DAF core, which can be configured and controlled with a Python script. The GUI basically just passes everything down to the core and thus works exactly the same as the core. Every object defined in the GUI is converted into a real Python object, the same way as it would be done in a Python script that defines the core. Thus, the **documentation of the core** is almost entirely relevant when using the GUI. If you feel like something explained in the GUI does not make sense or is missing something, please refer to the *core documentation*.

GUI structure

The GUI consists of:

- Connection section:
  - START / STOP for starting/stopping or connecting/disconnecting from the core.
  - Connection type selection (local or *remote*).
- Tabs:
  - Optional modules tab - Where you can install optional functionallity.
  - Schema definition tab - Where you can statically (as a template) define accounts, guilds, messages & type of logging:
    - Accounts - Section for defining your accounts (and guilds and messages).
    - Logging - Section for defining the logging manager used and the detail of the trace (printouts).
    - "Schema" menu button - Allows save or load of GUI data and generation of a Python script which will advertise defined data without a GUI. The script interacts directly with DAF core.
  - Live view - Manipulating running accounts, guilds, messages, etc..
  - Output tab for displaying the DAF core's printouts (eg. message removed, guild removed, started, stopped, ...),

- Analytics tab for tracking sent messages and invite links:

  This consists of 2 sub tabs, where the first one is for messages and the second one for invite links and each tab has 2 distinct sections:

  * Logs - Used to view the actual data stored inside a database.
  * Counts - Table that can show basic statistics related to the logs.

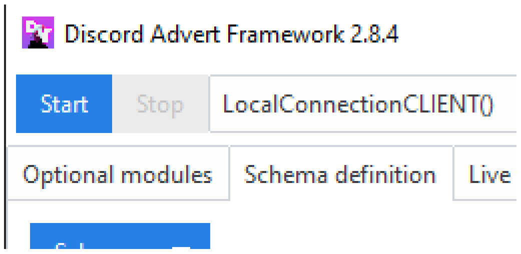
- About tab (short information on the project).

Framework control (GUI)

Run control

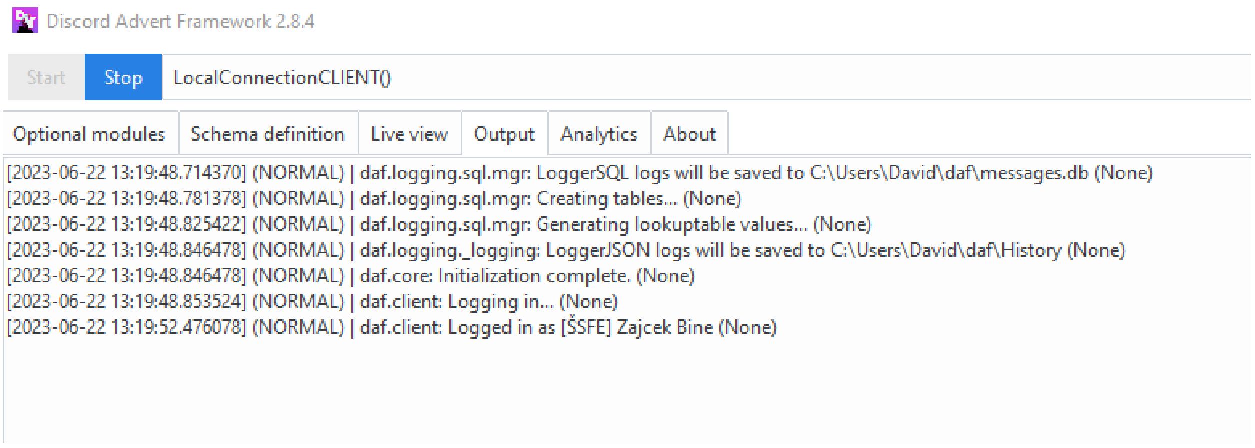
The framework can be from the GUI controlled with the 2 buttons located on the top left corner of the GUI.

These buttons are called **START** and **STOP** respectively.



Start button will, like the name suggest, start the framework (or connect to a remote server, see *Remote control (GUI)*). If the **Load all at start** checkbox inside *Schema definition* tab, Accounts frame, is checked, it will also load all of the defined account templates into DAF. Additionally the selected logger inside the Logging frame is set to be used to log messages and invites links and it cannot be changed unless the framework is restarted (the same goes for remote).

Inside the *Output* tab, the core's output can be viewed reporting any runtime errors.



```
[2023-06-22 13:19:48.714370] (NORMAL) | daf.logging.sql.mgr: LoggerSQL logs will be saved to C:\Users\David\daf\messages.db (None)
[2023-06-22 13:19:48.781378] (NORMAL) | daf.logging.sql.mgr: Creating tables... (None)
[2023-06-22 13:19:48.825422] (NORMAL) | daf.logging.sql.mgr: Generating lookuptable values... (None)
[2023-06-22 13:19:48.846478] (NORMAL) | daf.logging._logging: LoggerJSON logs will be saved to C:\Users\David\daf\History (None)
[2023-06-22 13:19:48.846478] (NORMAL) | daf.core: Initialization complete. (None)
[2023-06-22 13:19:48.853524] (NORMAL) | daf.client: Logging in... (None)
[2023-06-22 13:19:52.476078] (NORMAL) | daf.client: Logged in as [ŠSFE] Zajcek Bine (None)
```

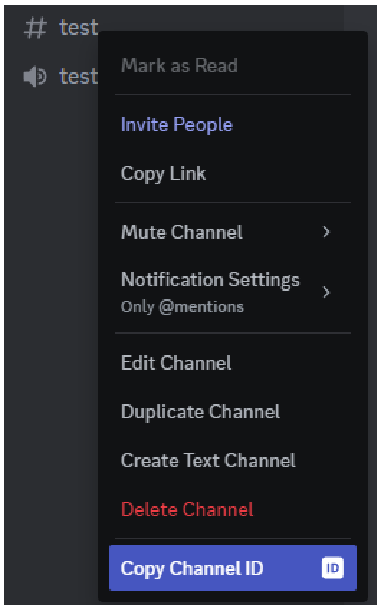
Account definition / schema tab (GUI)

The *Schema* tab allows users to define a fixed schema **template** that can then be saved (loaded) to (from) file. or converted into a Python `.py` script that runs exactly the same as it would inside the GUI.

Some **important terms** that users need to know if they wish to define objects:

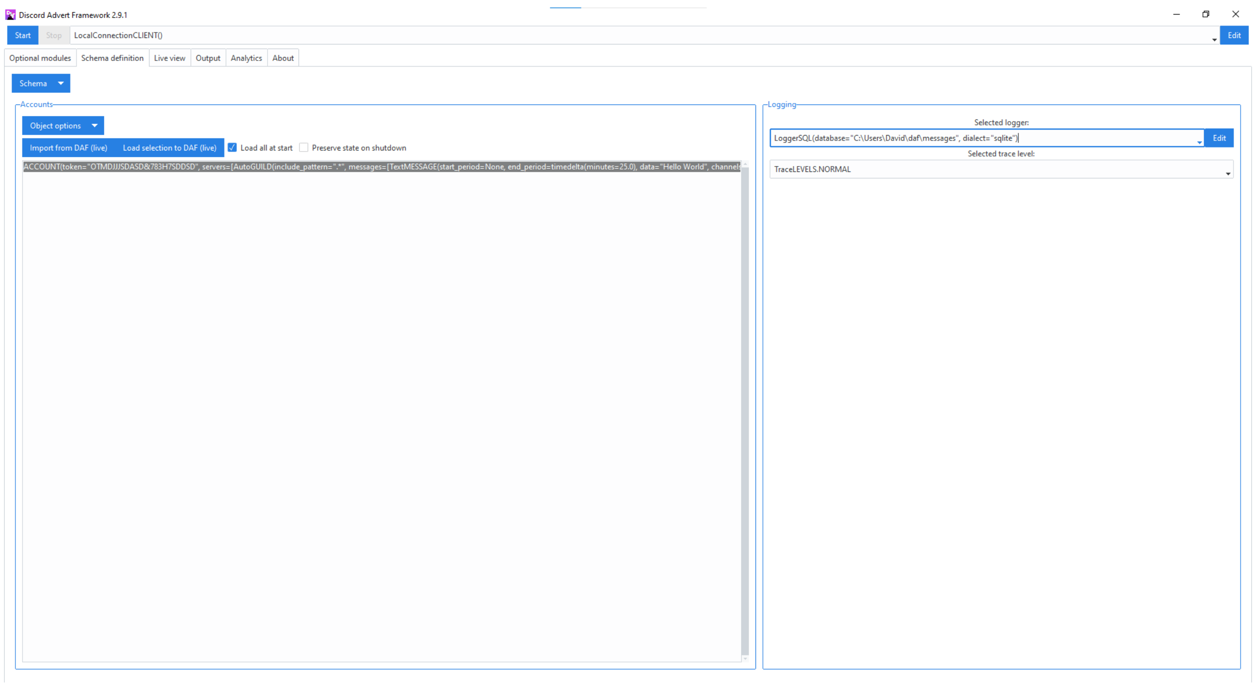
Token
Refers to the Discord account token, which can be obtained for bots though the developer tab or though a browser for user accounts.

Snowflake (ID)
The snowflake ID is Discord's unique identifier that each user, channel, guild, etc. has. It is never duplicated and it only represents a single Discord object. It is often needed inside DAF. To obtain it, first enable developer mode. Then you can right click on the wanted resource (eg. channel) and left click on *Copy ID*.



In the schema tab we can define:

1. Accounts
2. Logging & tracing
3. Connection manager, however this is not inside the schema tab but rather on the top of the GUI.



LocalConnectionCLIENT()

[✓] Load all at start

[ ] Preserve state on shutdown

```
ACCOUNT(token="OTMDJJJSDASD&783H7SDDSD", servers=[AutoGUILD(include_pattern=".*", messages=[TextMESSAGE(start_period=None, end_period=timedelta(minutes=25.0), data="Hello World", channels
```

Selected logger: LoggerSQL(database="C:\Users\David\daf\messages", dialect="sqlite")

Selected trace level: TraceLEVELS.NORMAL

Defining ACCOUNT objects

We can define *ACCOUNT* objects by clicking on *Object options* -> *New ACCOUNT*. This opens a new object definition window.

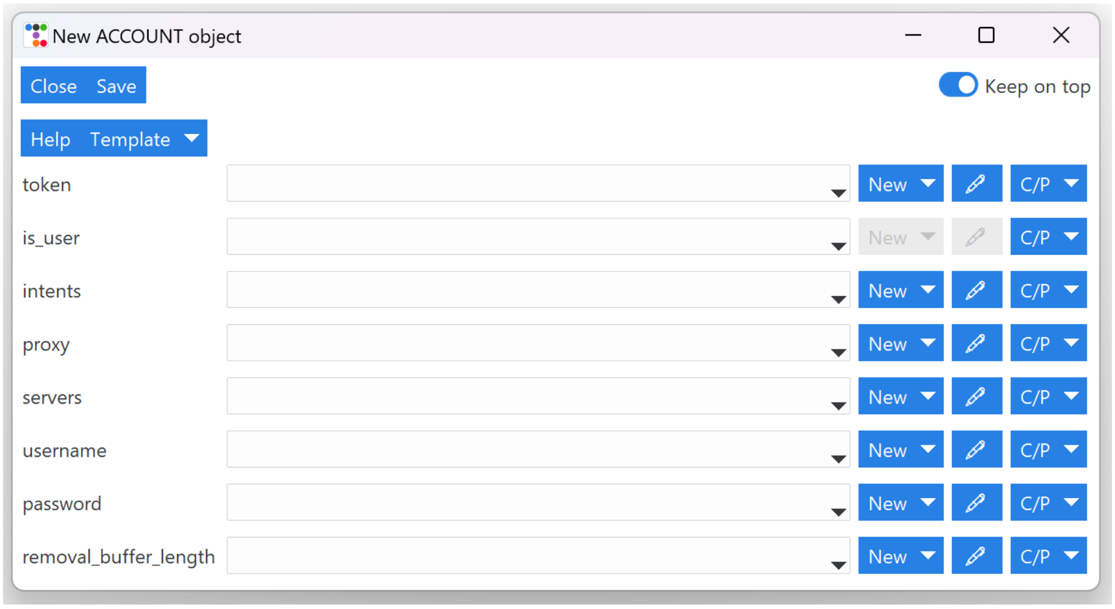


Fig. 5.1: Object definition window

Basic information about the object definition window

In the toolbar (top) we can observe 3 buttons and one toggle. The **Close** button closes the window and asks the user if they want to save the object to the previous widget, while the **Save** does the same thing except it saves the object without user confirmation. The toggle **Keep on top** will prevent other windows from covering the definition window.

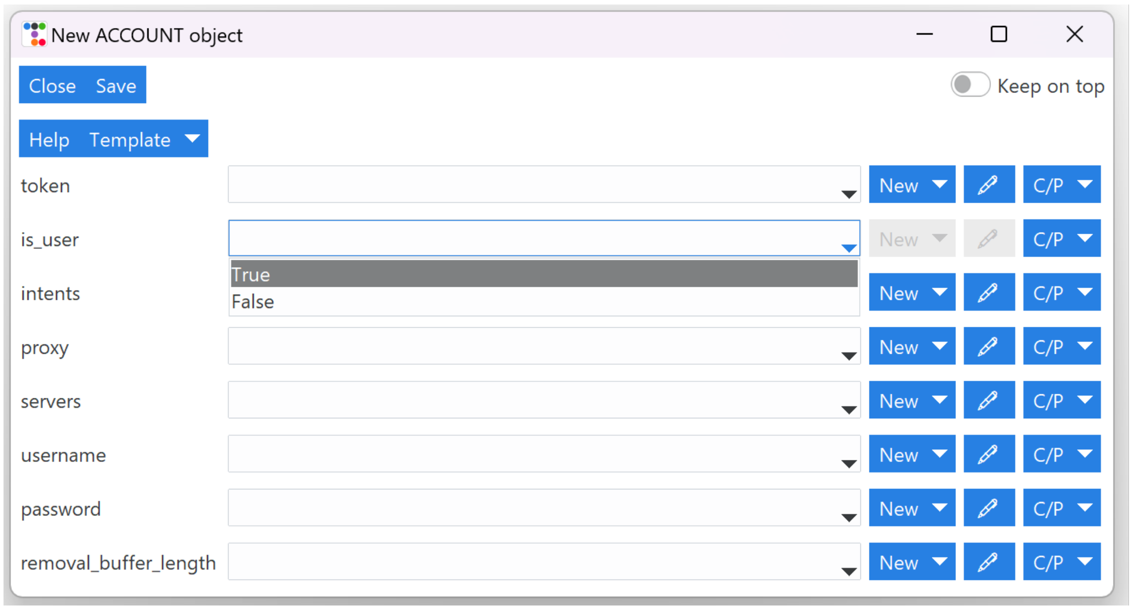
The **Help** button opens up the documentation and searches for the corresponding object, in our case, the `ACCOUNT` object. You can use this button to gain knowledge about what each parameter means.

When defining structured data there is an additional **Template** button which allows users to save (or load) the current parameters to (from) a JSON file. This is simillar to *Schema backup & Script generation (GUI)* except it only backups the current object.

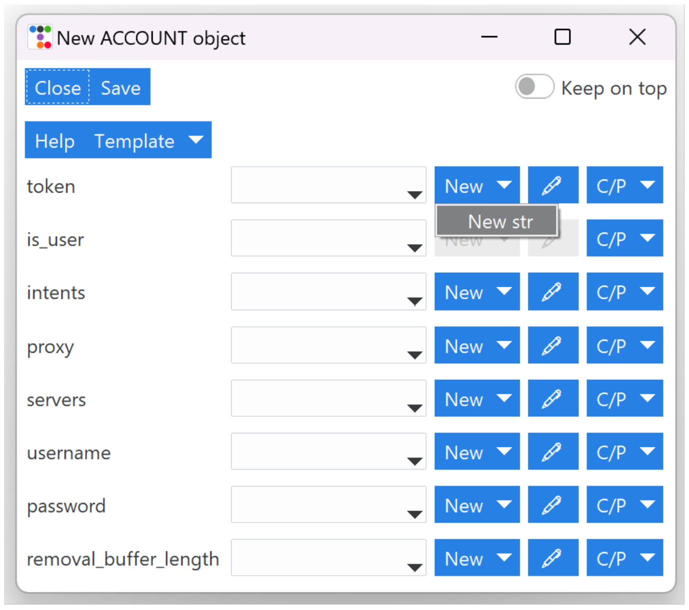
Note: Some data types will have additional widgets, such as Color Picker or Datetime select.

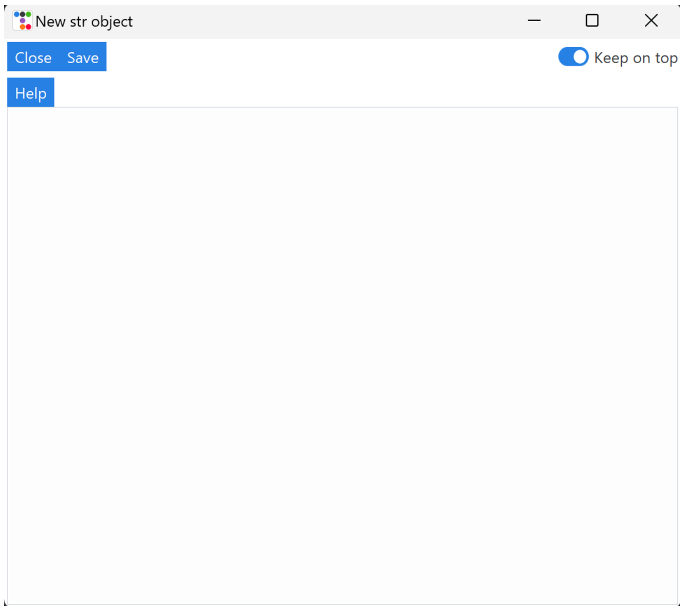
Depending on the datatype each parameter accepts, we can either:

1. Select a value from a predefined list by clicking the little arrow in the dropdown menu:

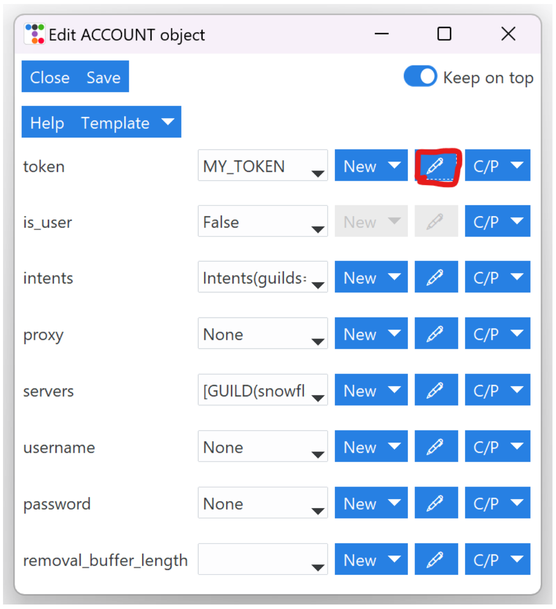


2. Create new value by clicking on the menu button *New* and then select the type you want to define. which will open another definition frame.

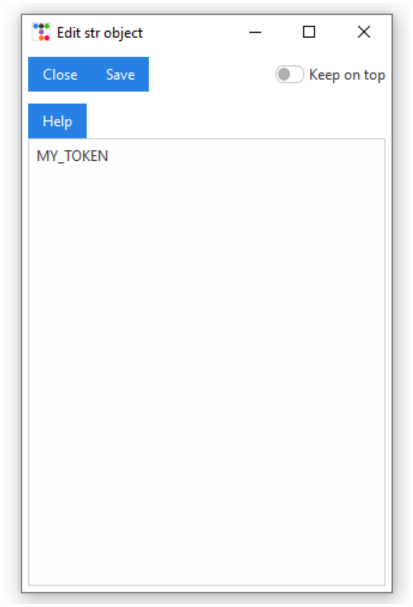




3. Edit current value by clicking on pencil button.



token	MY_TOKEN	New		C/P
is_user	False	New		C/P
intents	Intents(guilds:	New		C/P
proxy	None	New		C/P
servers	[GUILD(snowfl	New		C/P
username	None	New		C/P
password	None	New		C/P
removal_buffer_length		New		C/P



MY_TOKEN

After we are satisfied with our definition, we can click *Save* to save the changes into the parent (previous) object.

Account definition

To define an account we can choose from various parameters, the important ones for this guide are:

1. `token` - The Discord account token, you can obtain this the following way:
   - BOT accounts - https://discord.com/developers/applications (select your app -> Bot -> Reset / Copy token)
   - USER accounts (self-bots) - https://youtu.be/YEgFvgg7ZPI
2. `is_user` - Tells the framework the above token type, this must be set to `True` if you want advertise using an user account (self-bot).
3. `servers` - A list of *`GUILDS`* and *`USERS`* messages will be sent to.

Note: To logging with **username** and **password** we can use the corresponding fields in the definition window.

Logging in with username and password happens though the browser and requires additional dependencies which can be installed with:

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

This is only available on desktop computers and cannot be eg. run on a linux server. It is recommended to obtain the user token instead unless additional features from the web modules are desired (see *Web browser (core)*)

After defining the token and other normal parameters, we can define the servers. Defining servers will open up a new definition frame which allows you to define multiple GUILD objects.

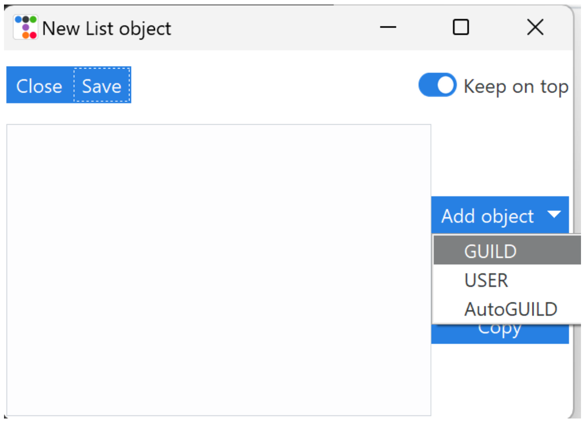


Fig. 5.2: New server definition window.

GUILD accepts parameters: snowflake, messages, logging and remove_after.

snowflake represents Discord's ID of the guild, messages a list of *TextMESSAGE* & *VoiceMESSAGE* objects, logging is a bool parameter which enables / disables logging of sent messages for this guild and remove_after parameter specifies the time or time delay for this guild to be auto removed from the list.

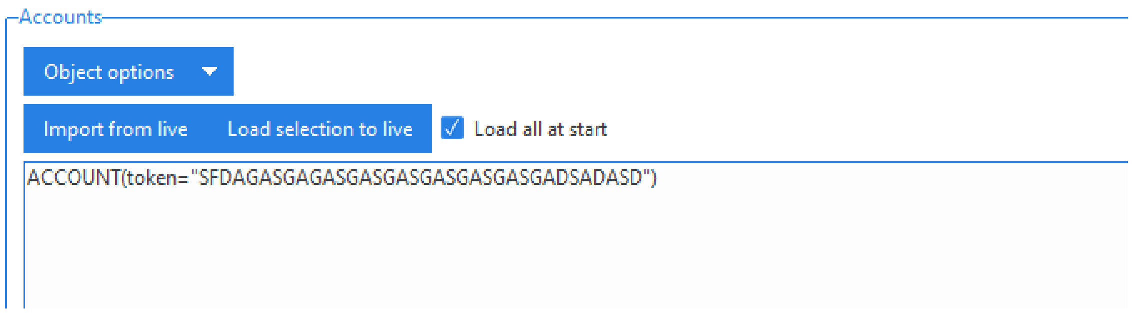
USER accepts the same parameters, except messages is a list of *DirectMESSAGE* objects.

For more information about the parameters and definition for other objects click the *Help* button or search for the object in *API reference* or read the *core guide*.

To define other objects (such as messages), please refer to the *Shilling list definition (core)* documentation, part of the DAF's core guide. When reading the core documentation, just define everything inside the GUI the same way as it is done in the example code.

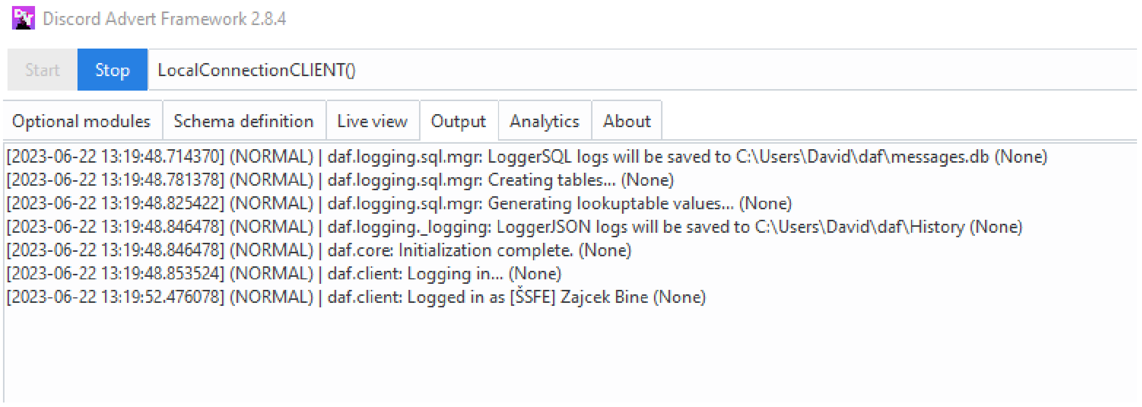
Successful account definition

After successful definition, we can observe a new account added to our accounts list.

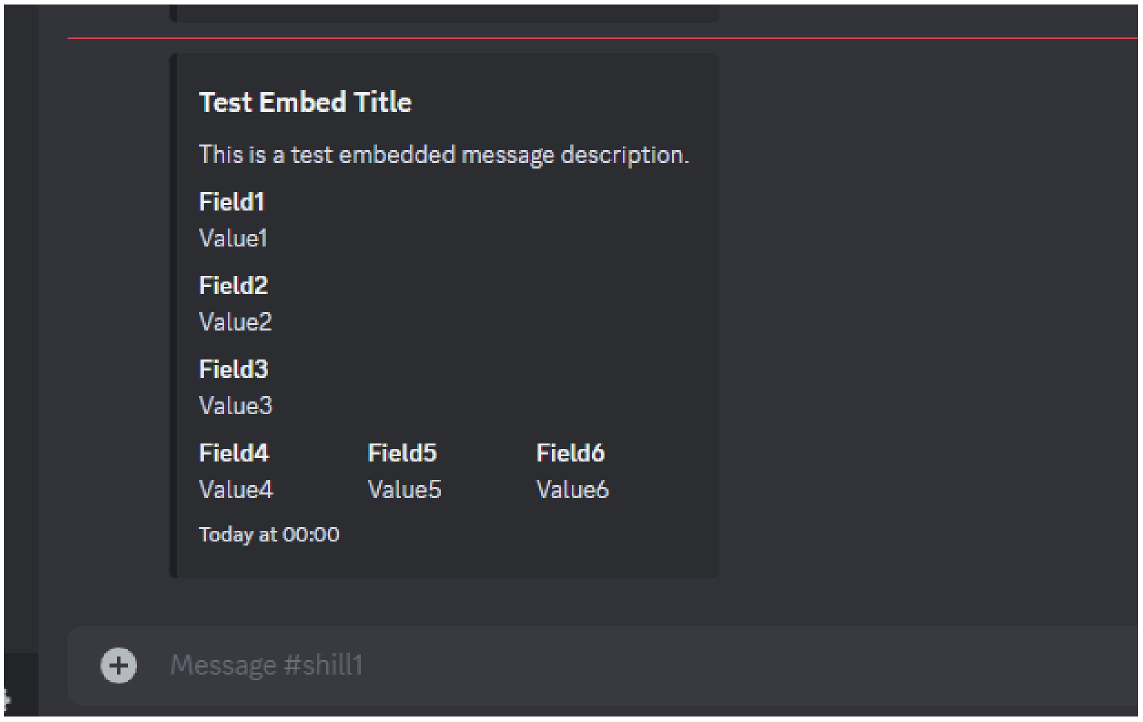


```
ACCOUNT(token="SFDAGASGAGASGASGASGASGASGASGADSADASD")
```

If we click the *Start* (on top of the main window), we can observe our account being logged-in and messages being sent to the defined guilds and channels.



```
[2023-06-22 13:19:48.714370] (NORMAL) | daf.logging.sql.mgr: LoggerSQL logs will be saved to C:\Users\David\daf\messages.db (None)
[2023-06-22 13:19:48.781378] (NORMAL) | daf.logging.sql.mgr: Creating tables... (None)
[2023-06-22 13:19:48.825422] (NORMAL) | daf.logging.sql.mgr: Generating lookuptable values... (None)
[2023-06-22 13:19:48.846478] (NORMAL) | daf.logging._logging: LoggerJSON logs will be saved to C:\Users\David\daf\History (None)
[2023-06-22 13:19:48.846478] (NORMAL) | daf.core: Initialization complete. (None)
[2023-06-22 13:19:48.853524] (NORMAL) | daf.client: Logging in... (None)
[2023-06-22 13:19:52.476078] (NORMAL) | daf.client: Logged in as [ŠSFE] Zajcek Bine (None)
```



Test Embed Title

This is a test embedded message description.

Field1
Value1

Field2
Value2

Field3
Value3

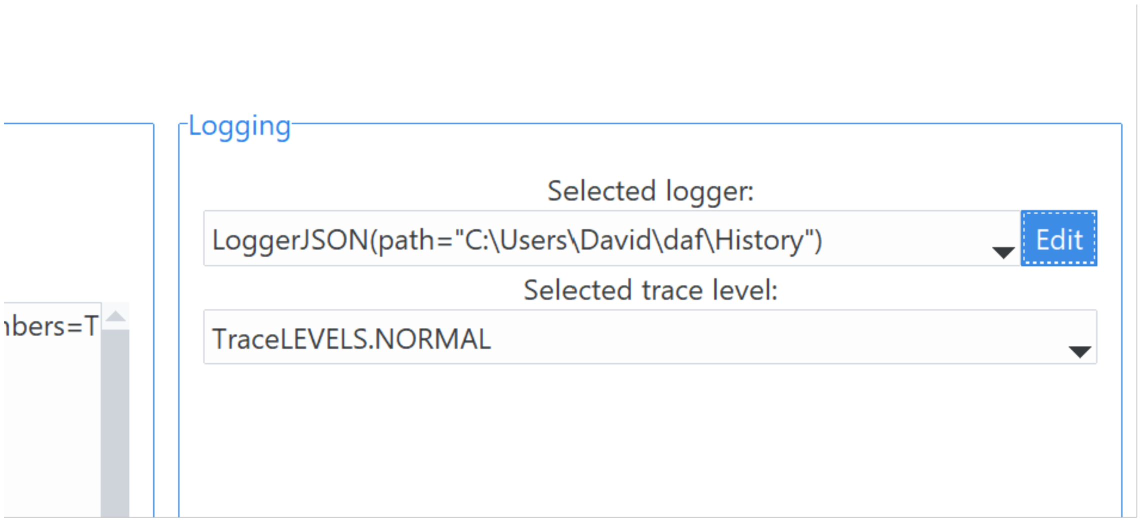
Field4	**Field5**	**Field6**
Value4	Value5	Value6

Today at 00:00

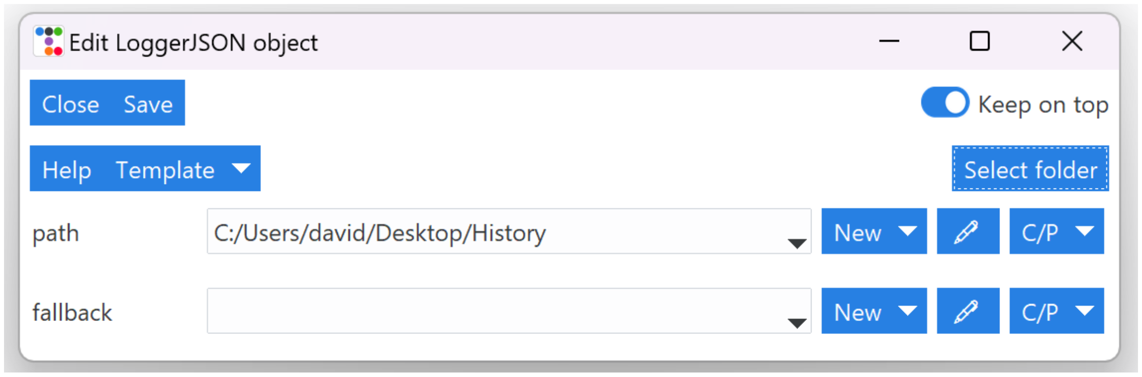
Logging definition (GUI)

We can also define what type of logging DAF will use to log sent messages and the detail of prints inside the *Logging* section of the *Schema definition* tab

To configure a logger we can select the wanted logger and click on the *Edit* button, located on the right side of the 1st dropdown menu.



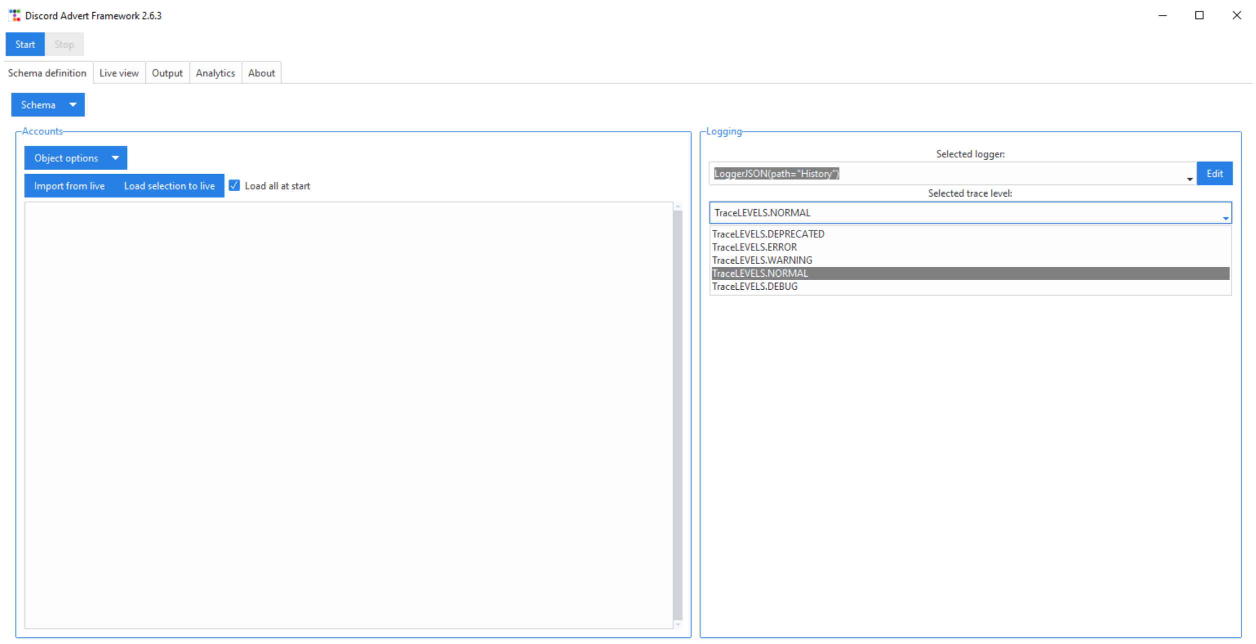
Selected logger:
LoggerJSON(path="C:\Users\David\daf\History")
Selected trace level:
TraceLEVELS.NORMAL



Edit LoggerJSON object
path C:/Users/david/Desktop/History
fallback

After clicking on *Save*, our logging manager is now defined and will be responsible for logging the data.

In the 2nd dropdown we can now select the debug / trace level. Value *DEPRECATED* will only show deprecation notices and is the least detailed trace configuration, while *DEBUG* will print all the information, including the debugging ones - it is considered the most detailed trace configuration.



NORMAL trace is recommended for most users.

For more information about logging refer to the core documentation - *Logging (core)*.

Loading schema into DAF (GUI)

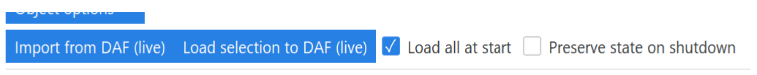
The *Load all at start* checkbox causes (when checked) the GUI to load all the accounts into DAF right after the *Start* button in the top left corner is pressed. If the checkbox is not checked, accounts can be loaded by selecting them in the list and then clicking on the *Load selection to live* button.

The *Preserve state on shutdown* checkbox sets the `save_to_file` parameter inside `run()` to True if checked or False if not checked. Basically this means that if the checkbox is checked, DAF will save the accounts list (and guilds, messages, ...) to a binary file on DAF shutdown and every 2 minutes to prevent data loss on force shutdown. When starting DAF again, the same list will be loaded from file into DAF.

State preservation is not really meant as a shilling feature where you can define the schema statically inside the GUI and save it to a JSON file, but it's meant to be used in case DAF will have accounts, guilds, messages dynamically added while it's running (added in *Live view*).

If *Import from live* is pressed, the GUI will copy the accounts loaded inside daf into our list.

Logger is automatically loaded at start and cannot be changed for a different logger without stopping the framework first.



Live view (GUI)

While the Schema tab allows user to pre-define a schema, the GUI also has a section called *Live view*.

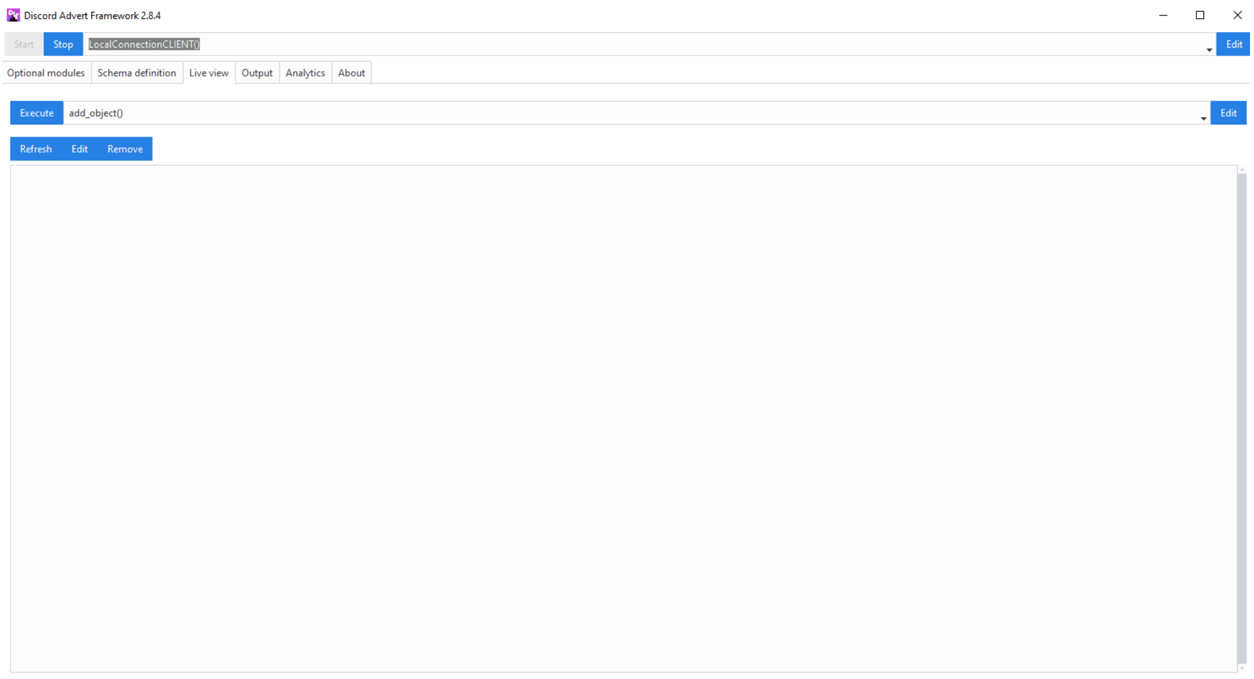
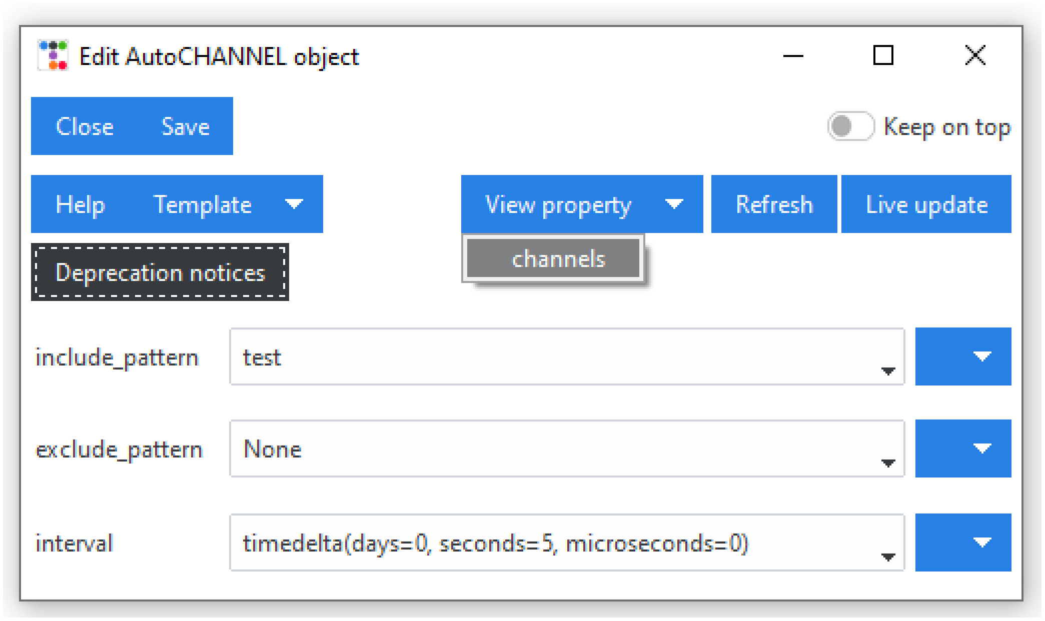


Fig. 5.3: Live view tab

Live view allows users to view and update objects that are currently loaded into the framework. It allows users to modify the original object parameters and then reinitialize the object to use those same parameters. This can be done by clicking on the **Update** button. Next to the *Update* button, there is a **Refresh** button which will reload the GUI with updated values. If an object does not have a *Update* button, that means the object is not directly supported for live modifications. In the latter case, users must click the *Save* button, until edit for an object, which supports live update, is found. **Clicking on Save does not reflect changes in the actual framework**, but just in the currently opened window.

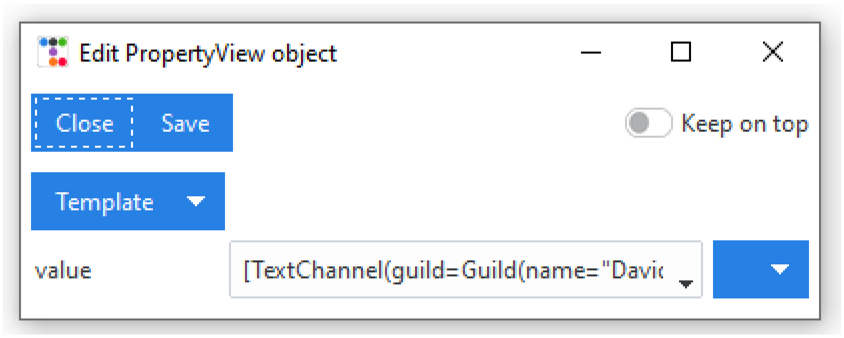
When editing / viewing a live object, there is also a **View property** menu button, which allows users to inspect additional object properties of the object running inside DAF. This can eg. be used to view the channels, that were found using `AutoCHANNEL` object.



include_pattern: test

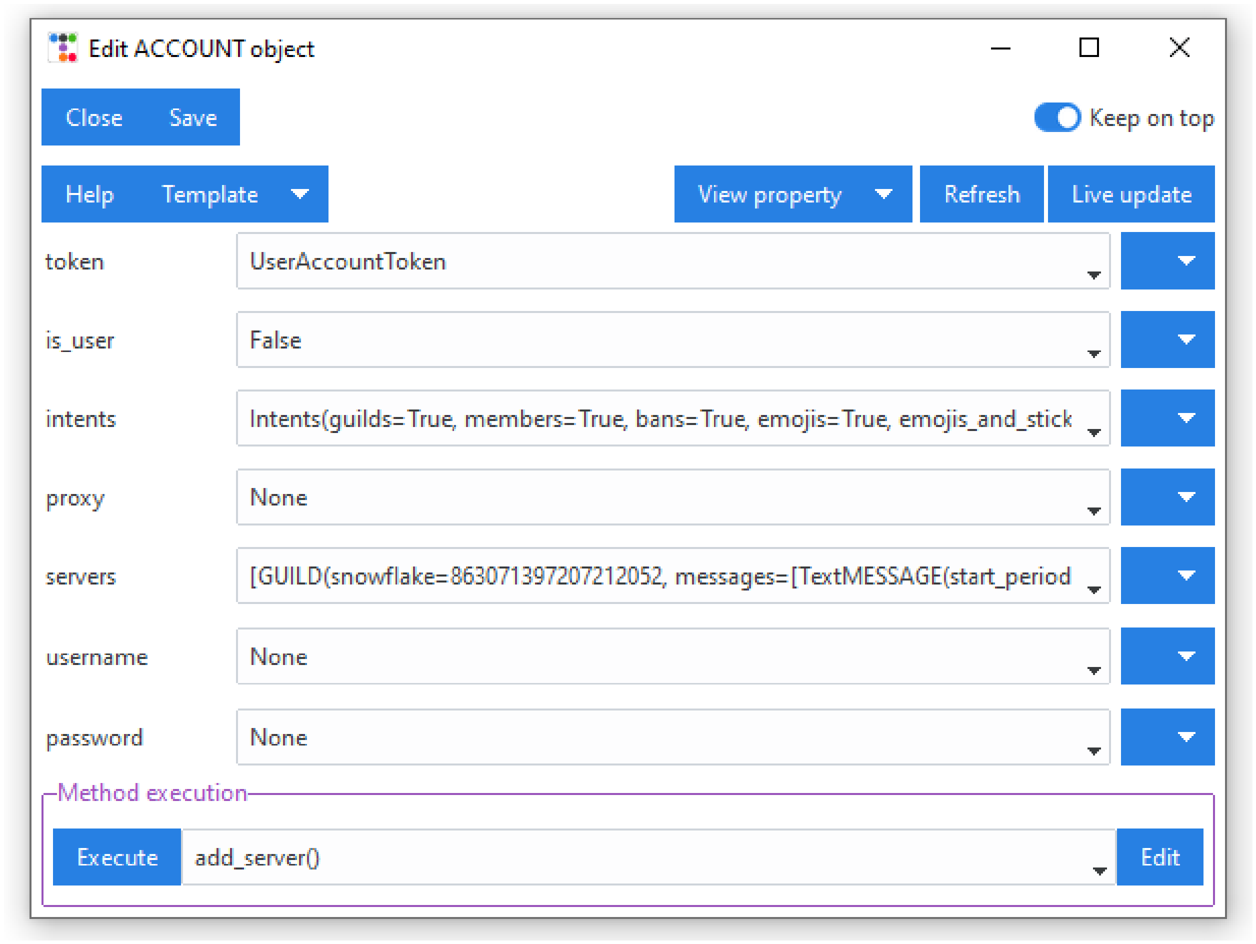
exclude_pattern: None

interval: timedelta(days=0, seconds=5, microseconds=0)



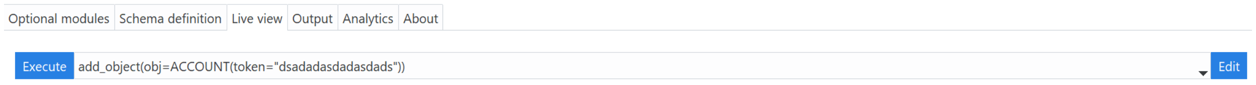
value: [TextChannel(guild=Guild(name="Davi

Some live objects like *ACCOUNT* also support **method execution**, where you can eg. add or remove servers without updating the entire object.



token: UserAccountToken
is_user: False
intents: Intents(guilds=True, members=True, bans=True, emojis=True, emojis_and_stick
proxy: None
servers: [GUILD(snowflake=863071397207212052, messages=[TextMESSAGE(start_period
username: None
password: None
add_server()

At the top of the *Live view* tab, there's also an *Execute* button with a dropdown menu. It allows you to define a new `ACCOUNT` object by clicking *Edit* and load the object into the framework directly. However it is recommended that accounts are defined inside the *Account definition / schema tab (GUI)* and loaded by clicking the *Load selection to live* button (see *Loading schema into DAF (GUI)*).



add_object(obj=ACCOUNT(token="dsadadasdadasdads"))

Schema backup & Script generation (GUI)

DAF's graphical interface allows us to define a schema and run it directly from the GUI.

In case our plan is to advertise to a lot of servers it wouldn't be practical if we had to re-define the schema every time we re-open the GUI. This is why DAF-GUI supports **schema backup and restore** and can additionally generate a core DAF Python script for shilling from the console (appropriate for servers) without needing the graphical interface.

The backups are saved into `JSON` files and contain definition of account objects (and it's members such as guilds, messages), logger list and selected logger & the selected trace configuration.

Schema backup and restore

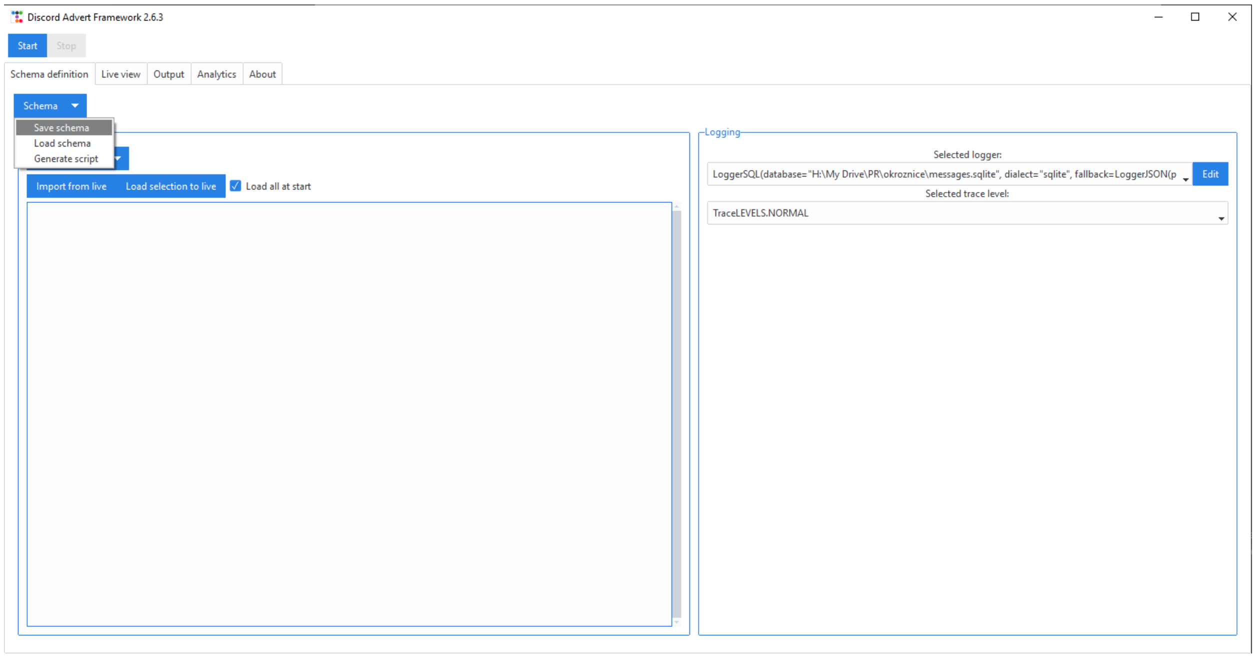
Schema backup

See also:

`Download example schema backup (schema.json)`

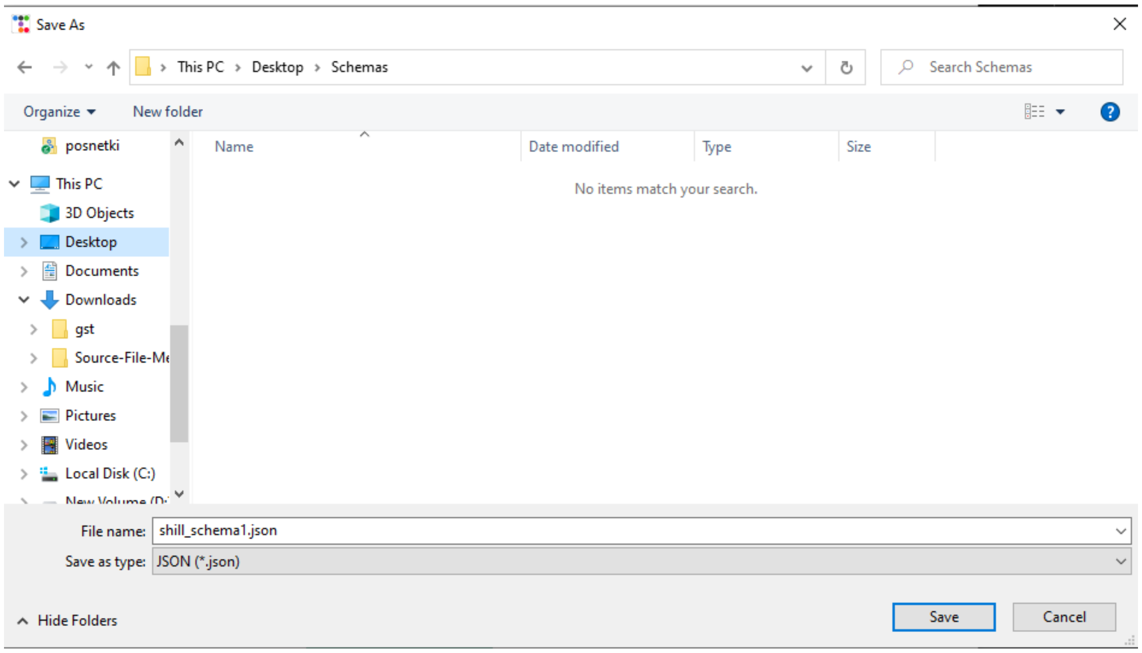
To save a schema after it has been defined, we can click on the *Schema* button, located in the top left corner of the *Schema definition* tab and then click the **Save schema** option.

We will be asked for a file location of the schema backup, where the GUI will save backup our schema.



LoggerSQL(database="H:\My Drive\PR\okroznice\messages.sqlite", dialect="sqlite", fallback=LoggerJSON(p

TraceLEVELS.NORMAL



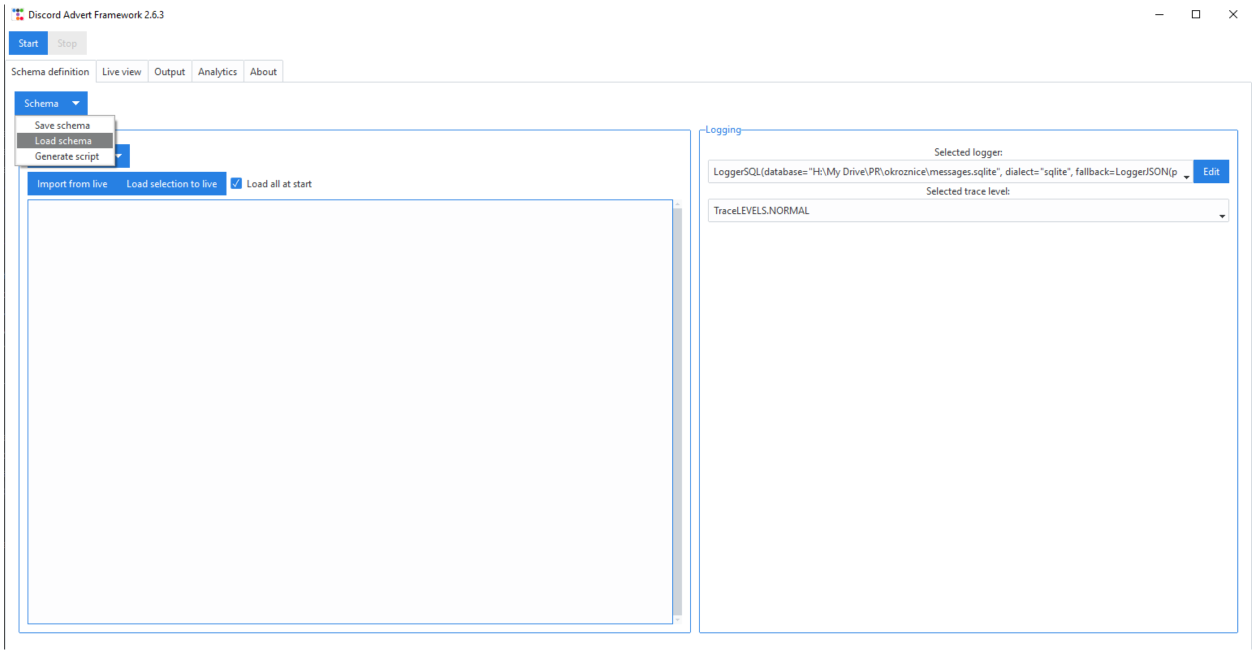
File name: shill_schema1.json

Save as type: JSON (*.json)

Schema restore

In case we want to restore a backed-up schema, we can click on the *Schema* button, located in the top left corner of the *Schema definition* tab and then click the **Load schema** option.

We will be asked for a file location of the schema backup, from which the GUI will load our schema.



Selected logger:
LoggerSQL(database="H:\My Drive\PR\okroznice\messages.sqlite", dialect="sqlite", fallback=LoggerJSON(p

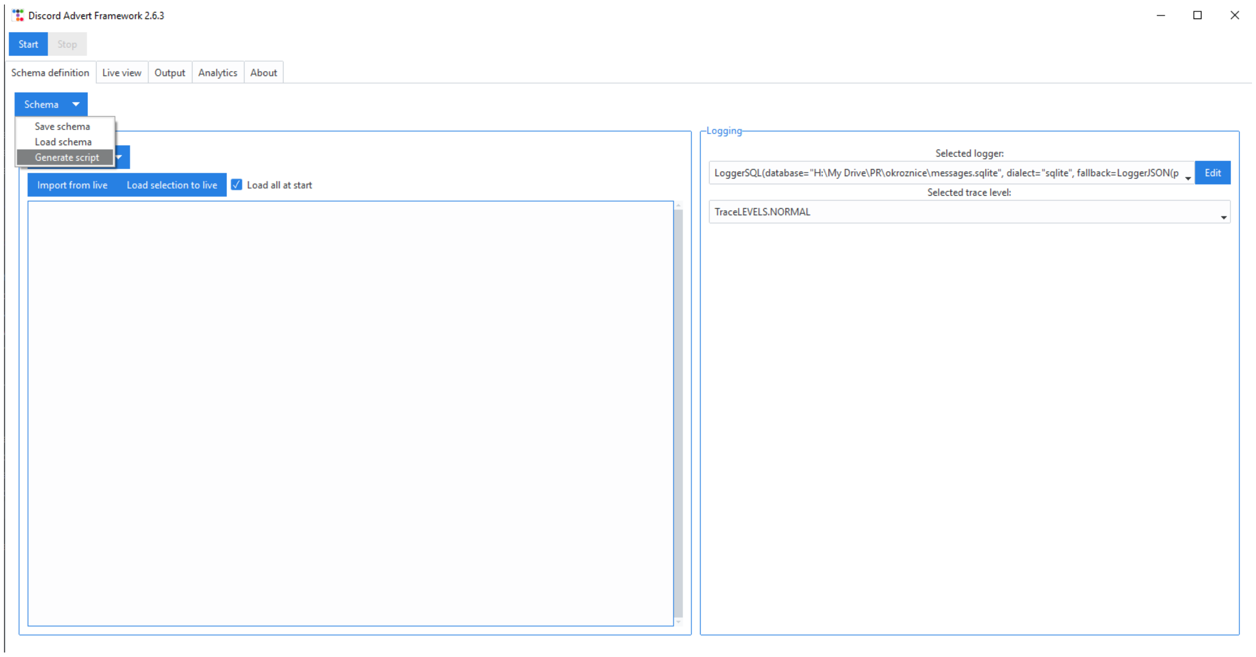
Selected trace level:
TraceLEVELS.NORMAL

Shilling script generation

Before the graphical interface, DAF could only be run from a Python script in which everything was defined. The graphical interface makes it easier to shill for those who don't know Python, but it also limits it's use to a a computer capable of displaying image.

Luckily DAF-GUI allows use to create a (Python) shilling script in the event that we want to run DAF on a server 24/7.

To generate a shilling script , we can click on the *Schema* button, located in the top left corner of the *Schema definition* tab and then click the **Generate script** option. This will open up a file dialog asking us where to save the shilling script. After we save the location our shilling script will be generated and we can run it by using the command `python <out script here>`.



Selected logger:
LoggerSQL(database="H:\My Drive\PR\okroznice\messages.sqlite", dialect="sqlite", fallback=LoggerJSON(p

Selected trace level:
TraceLEVELS.NORMAL

Here is an example of the shilling script generated from the above option.

Listing 5.1: shill_script.py - Example shilling script generated from the GUI.

```
"""
Automatically generated file for Discord Advertisement Framework v2.6.0.
This can be run eg. 24/7 on a server without graphical interface.

The file has the required classes and functions imported, then the logger is defined and␣
↪the
accounts list is defined.

At the bottom of the file the framework is then started with the run function.
"""

# Import the necessary items
from daf.logging._logging import LoggerJSON
from daf.logging.tracing import TraceLEVELS
from daf.client import ACCOUNT
from datetime import timedelta
from datetime import datetime
from _discord.embeds import Embed
from _discord.embeds import EmbedField
from daf.message.text_based import TextMESSAGE
from daf.guild import GUILD
from daf.message.base import AutoCHANNEL

import daf

# Define the logger
logger = LoggerJSON(path="./OutputPath/")
```

(continues on next page)

(continued from previous page)

```
# Defined accounts
accounts = [
    ACCOUNT(
        token="OTM2NzM5NDMxMDczODY1Nzg5.GLSf5S.u-KirCcieqFVZHdFItvrd0S6XFB-sKj-EMb298",
        is_user=False,
        servers=[
            GUILD(
                snowflake=863071397207212052,
                messages=[
                    TextMESSAGE(
                        start_period=None,
                        end_period=timedelta(
                            seconds=10.0,
                        ),
                        data=Embed(
                            title="Test Embed Title",
                            type="rich",
                            description="This is a test embedded message description.",
                            timestamp=datetime(
                                year=2023,
                                month=4,
                                day=2,
                            ),
                            fields=[
                                EmbedField(
                                    name="Field1",
                                    value="Value1",
                                ),
                                EmbedField(
                                    name="Field2",
                                    value="Value2",
                                ),
                            ],
                        ),
                        channels=AutoCHANNEL(
                            include_pattern="shill",
                            interval=timedelta(
                                seconds=5.0,
                            ),
                        ),
                        mode="send",
                        remove_after=datetime(
                            year=2025,
                            month=1,
                            day=1,
                        ),
                    ),
                ],
                logging=True,
                remove_after=datetime(
                    year=2023,
                    month=4,
```

(continues on next page)

(continued from previous page)

```
                    day=26,
                ),
            ),
        ],
    ),
]


# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL
)
```

We can run the above file with the following command:

```
$ python shill_script.py
```

and it will produce the following output while it is running:

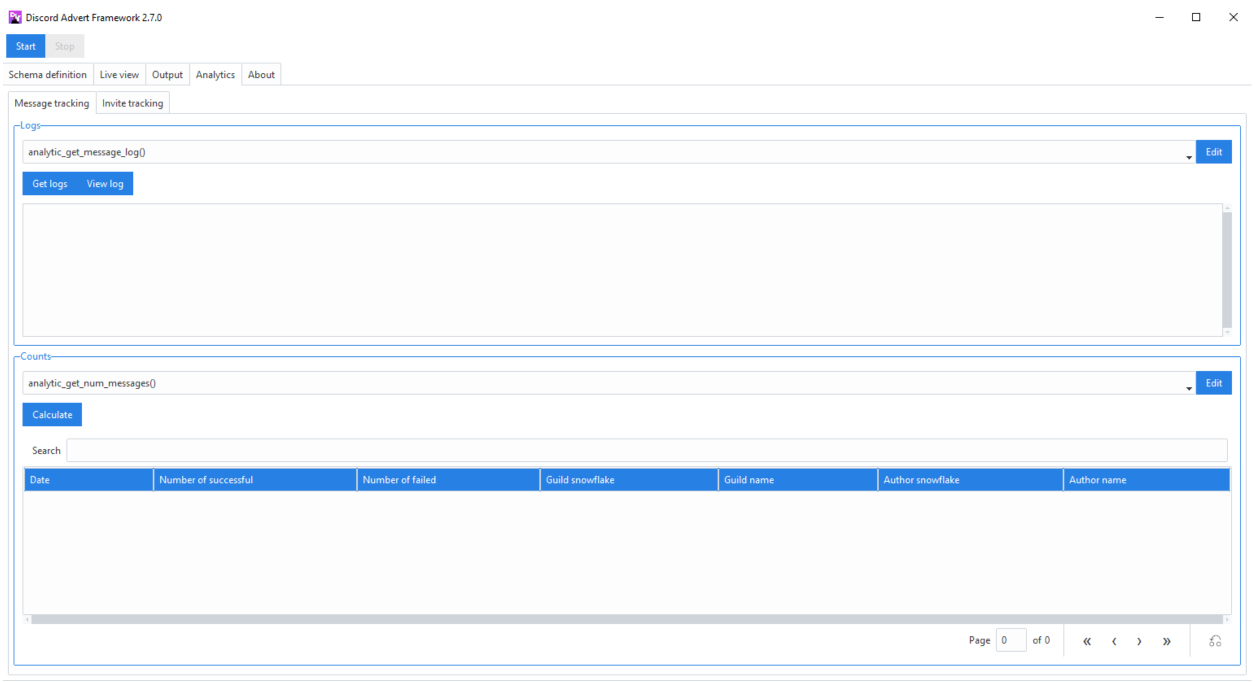
```
[2023-04-02 20:19:33.269412] (NORMAL) | daf.client: Logging in... (None)
[2023-04-02 20:19:36.016167] (NORMAL) | daf.client: Logged in as Aproksimacka (None)
[2023-04-02 20:19:36.016167] (NORMAL) | daf.core: Initialization complete. (None)
```

Note: The above script will run exactly the same as it would run inside the graphical interface. Graphical interface doesn't add any functionality it self to the Discord Advertisement Framework it self, it just makes it easier to use, which means that everything that happens in the GUI, can also happen in the core library (except schema backup / restore).

Analytics (GUI)

Changed in version 2.7: Added invite link tracking.

Caution: Analytics are currently **only** available if SQL is used for logging (*LoggerSQL*).



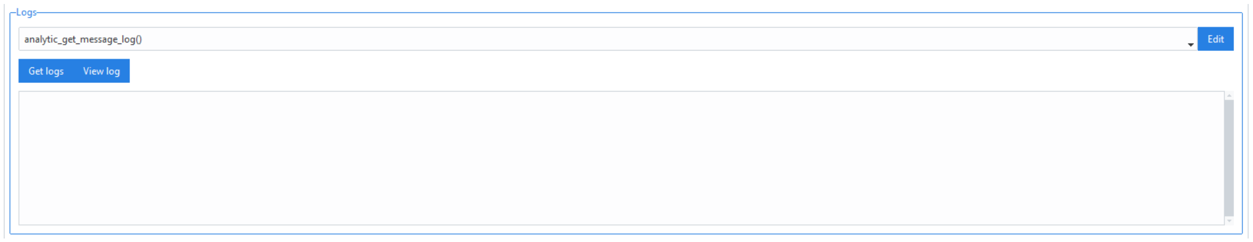
The *Analytics* tab allows users to get insight into sent messages and reached users. It is on the top-level split into 2 categories, which are *Message logging* and *Invite link tracking*. For both, a sub-tab exists under the *Analytics* tab.

Message logging

The message part of analytics is split into 2 sub-sections. These are *Logs frame (message)* and *Counts frame (message)*
.

Logs frame (message)

The logs frame allows retrieval of message logs and their display though the object edit window.

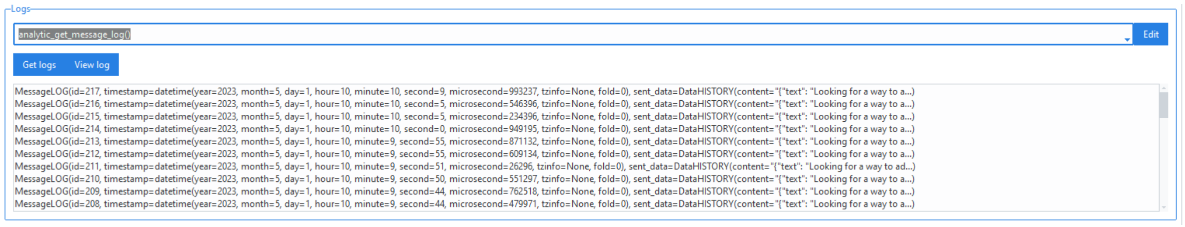


The frame contains a dropdown w/ an edit button for defining parameters and 2 buttons.

The edit button will open up an object edit window where you can define your query parameters. After you click save, the parameters will get saved into the dropdown menu. When using the *Get logs* button, these parameters will be considered. Use *Help* inside the definition window to get insight on the parameters.

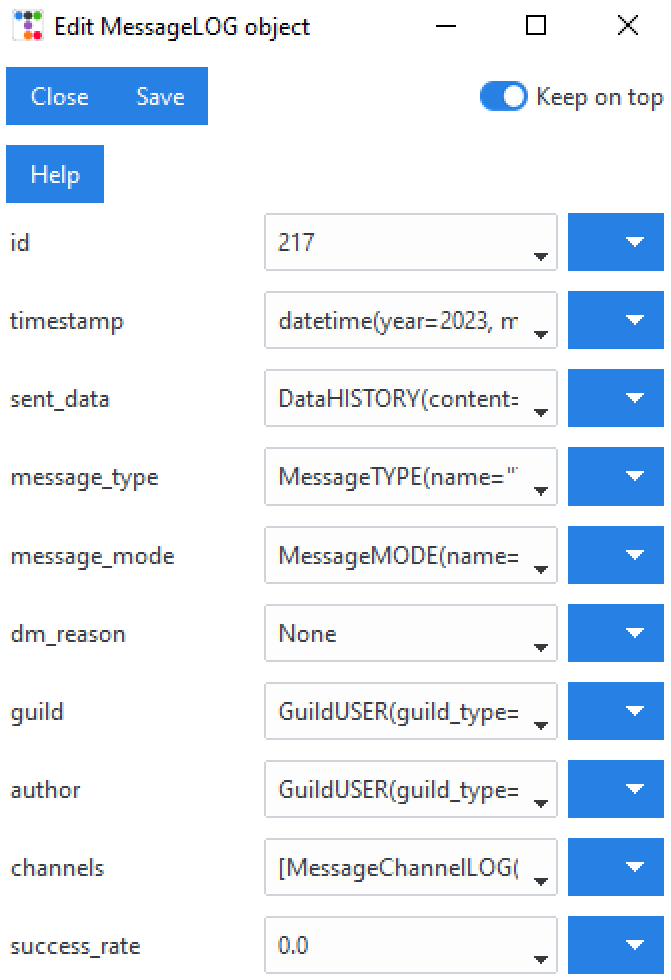
Buttons:

1. Get logs - retrieves the logs from the database and inserts them into the list at the bottom,



```
MessageLOG(id=217, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=10, second=9, microsecond=993237, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=216, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=10, second=5, microsecond=546396, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=215, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=10, second=5, microsecond=234396, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=214, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=10, second=0, microsecond=949195, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=213, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=9, second=55, microsecond=871132, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=212, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=9, second=55, microsecond=609134, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=211, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=9, second=51, microsecond=26296, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to ad...)
MessageLOG(id=210, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=9, second=50, microsecond=551297, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=209, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=9, second=44, microsecond=762518, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
MessageLOG(id=208, timestamp=datetime(year=2023, month=5, day=1, hour=10, minute=9, second=44, microsecond=479971, tzinfo=None, fold=0), sent_data=DataHISTORY(content="{"text": "Looking for a way to a...)
```

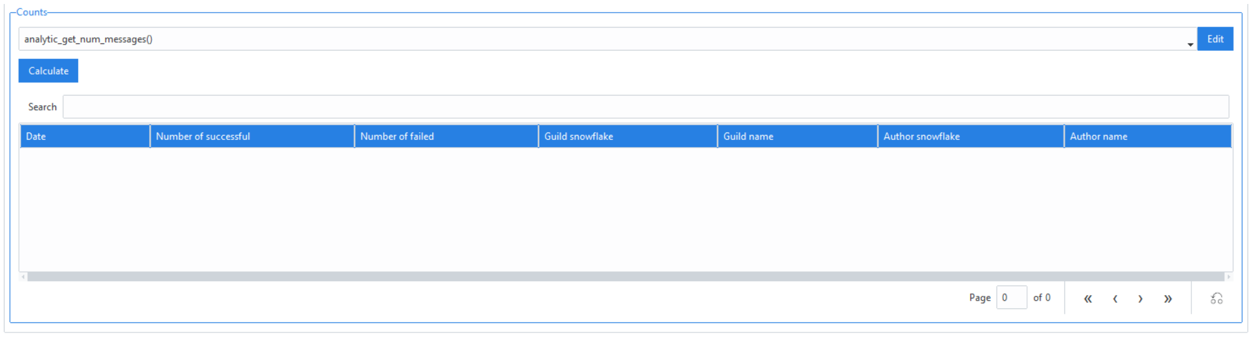
2. View log - opens an object edit window which can be used to inspect the log's content (read-only)



id	217
timestamp	datetime(year=2023, m
sent_data	DataHISTORY(content=
message_type	MessageTYPE(name="
message_mode	MessageMODE(name=
dm_reason	None
guild	GuildUSER(guild_type=
author	GuildUSER(guild_type=
channels	[MessageChannelLOG(
success_rate	0.0

Counts frame (message)

Counts frame can be used to count the amount of successful / failed message attempts into a specific guild from specific author on the selected day / month / year.

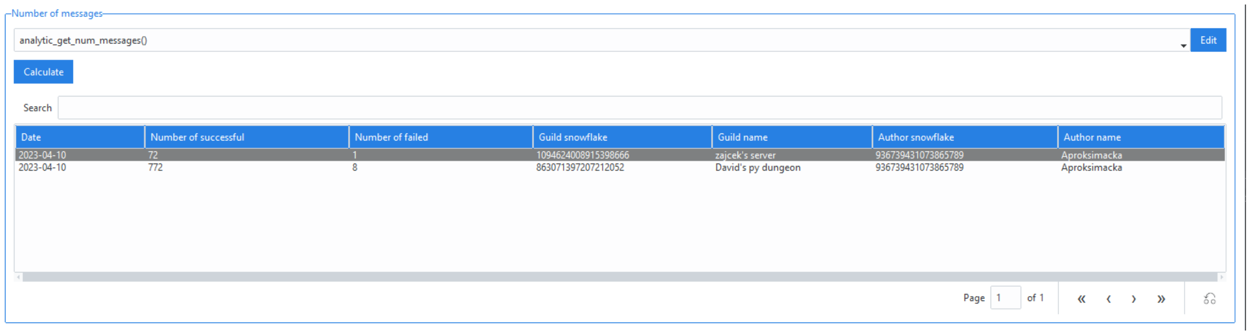


Date	Number of successful	Number of failed	Guild snowflake	Guild name	Author snowflake	Author name

The frame contains one dropdown w/ edit button which can be used to configure the query and one *Calculate* button.

Clicking on *Edit* will open up a object definition window where you can define the parameters. Use *Help* inside the definition window to get insight on the parameters.

Clicking on *Calculate* does the SQL analysis and inserts the result into the table below.



Date	Number of successful	Number of failed	Guild snowflake	Guild name	Author snowflake	Author name
2023-04-10	72	1	1094624008915398666	zajcek's server	936739431073865789	Aproksimacka
2023-04-10	772	8	863071397207212052	David's py dungeon	936739431073865789	Aproksimacka

Invite link tracking

The invite tracking is split into 2 sub-sections. These are *Logs frame (invites)* and *Counts frame (invites)* .

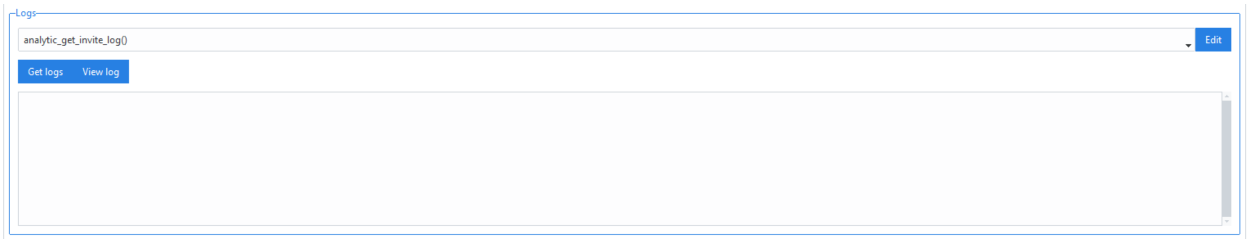
Warning: To track invite links, the Members intent (event setting) is needed. To use invite link tracking, users need to enable the privileged intent 'SERVER MEMBERS INTENT' inside the Discord developer portal and also set the `members` intent to True inside the `intents` parameter of *ACCOUNT*.

Invites intent is also needed. Enable it by setting `invites` to True inside the `intents` parameter of *ACCOUNT*.

Invite link tracking is **bot account** only and does not work on user accounts.

Logs frame (invites)

The logs frame allows retrieval of member joins with specific invite link and their display though the object edit window.



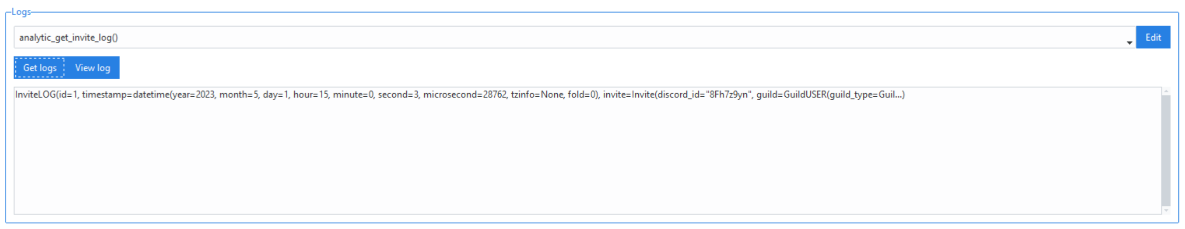
```
analytic_get_invite_log()
```

The frame contains a dropdown w/ an edit button for defining parameters and 2 buttons.

The edit button will open up an object edit window where you can define your query parameters. After you click save, the parameters will get saved into the dropdown menu. When using the *Get logs* button, these parameters will be considered. Use *Help* inside the definition window to get insight on the parameters.

Buttons:

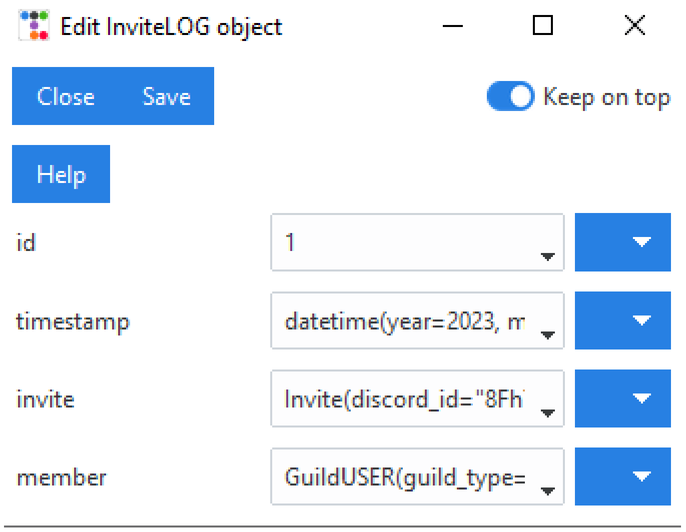
1. Get logs - retrieves the logs from the database and inserts them into the list at the bottom,



```
analytic_get_invite_log()
```

```
InviteLOG(id=1, timestamp=datetime(year=2023, month=5, day=1, hour=15, minute=0, second=3, microsecond=28762, tzinfo=None, fold=0), invite=Invite(discord_id="8Fh7z9yn", guild=GuildUSER(guild_type=Guil...)
```

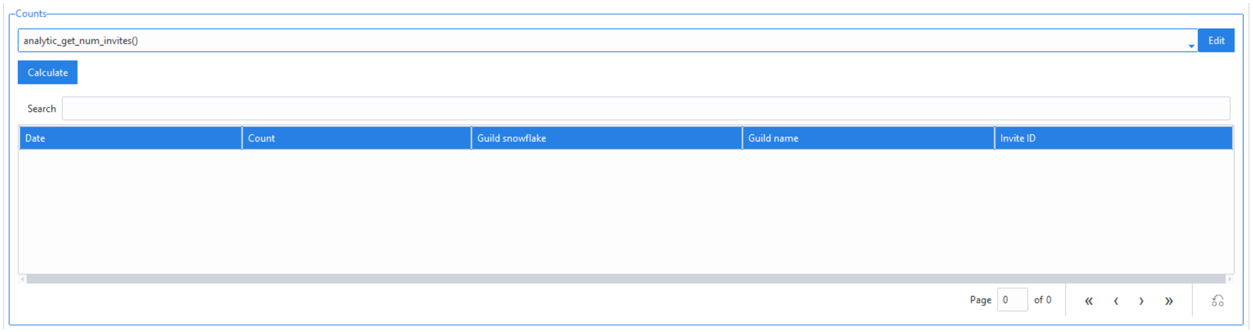
2. View log - opens an object edit window which can be used to inspect the log's content (read-only)



id	1
timestamp	datetime(year=2023, m
invite	Invite(discord_id="8Fh
member	GuildUSER(guild_type=

Counts frame (invites)

Counts frame can be used to count the amount of successful / failed message attempts into a specific guild from specific author on the selected day / month / year.

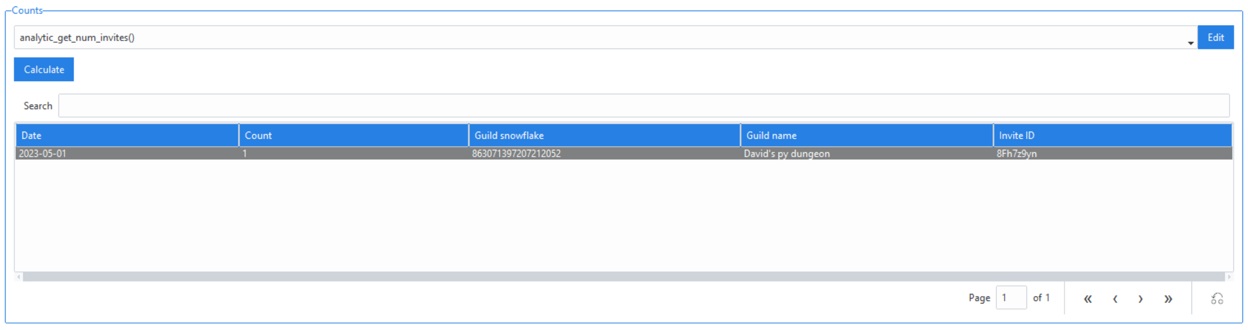


Date	Count	Guild snowflake	Guild name	Invite ID

The frame contains one dropdown w/ edit button which can be used to configure the query and one *Calculate* button.

Clicking on *Edit* will open up a object definition window where you can define the parameters. Use *Help* inside the definition window to get insight on the parameters.

Clicking on *Calculate* does the SQL analysis and inserts the result into the table below.



Date	Count	Guild snowflake	Guild name	Invite ID
2023-05-01	1	863071397207212052	David's py dungeon	8Fh7z9yn

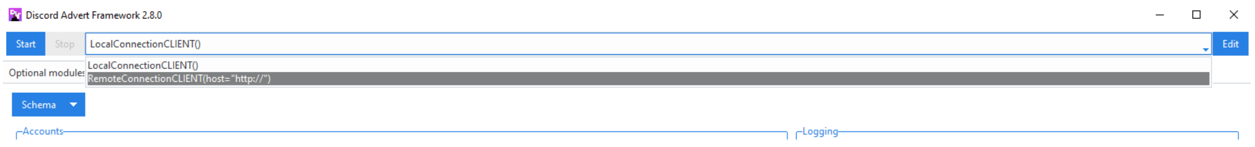
Remote control (GUI)

New in version 2.8.

See also:

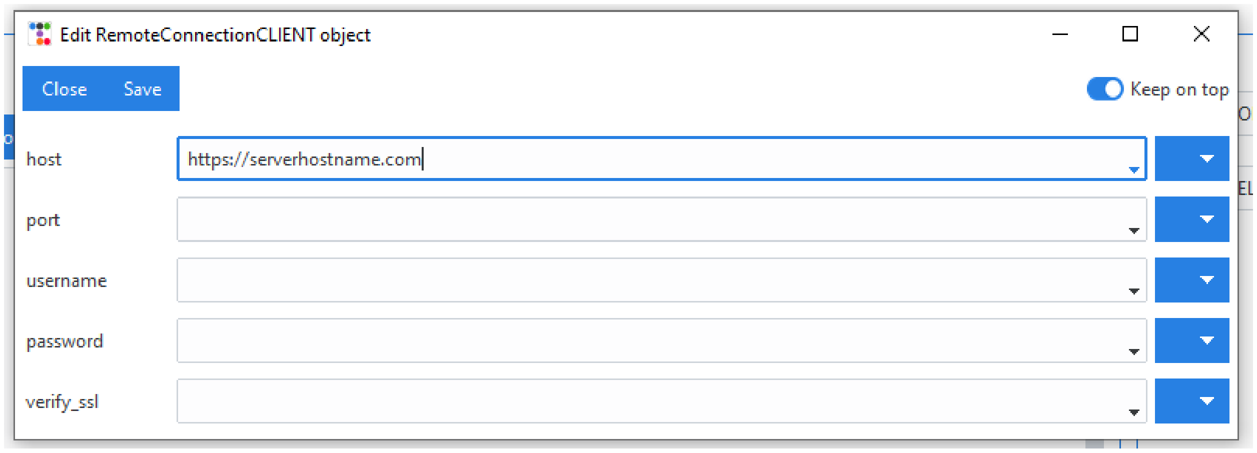
This section describes how to set up the GUI to connect to a server. To find information about setting up the actual server see *Remote control (core)*.

DAF can run either locally or it can connect to a remote server that is running the DAF core. Users can choose between these two options by changing the connection client as shown in picture:



Using `LocalConnectionCLIENT` will start DAF locally and anything the users do will be done locally. Using *`RemoteConnectionCLIENT`* will connect to a remote server and anything the users do including adding / removing accounts, retrieving logs, etc. will be done through a HTTP API which can also be HTTPS (recommended) if desired.

If the *Edit* button is clicked (in the top right corner) and *`RemoteConnectionCLIENT`* is selected, a new window will be opened, which allows customization of connection parameters.


Edit RemoteConnectionCLIENT object

host https://serverhostname.com
port
username
password
verify_ssl

class daf_gui.connector.**RemoteConnectionCLIENT**(*host: str*, *port: int = 80*, *username: str | None = None*, *password: str | None = None*, *verify_ssl: bool | None = True*)

Client used for connecting to DAF running on a remote server though the HTTP protocol. The connection is based fully on request from the GUI (this client).

Parameters

- **host** (*str*) – The URL / IP of the host.
- **port** (*Optional[int]*) – The HTTP port of the host. Defaults to 80.
- **username** (*Optional[str]*) – The username to login with.
- **password** (*Optional[str].*) – The password to login with.
- **verify_ssl** (*Optional[bool]*) – Defaults to True. If True, connection will be refused when the certificate does not match the host name.

See also:

When *generating the DAF core script*, remote access will also be configured if the `RemoteConnectionCLIENT` is selected.

Clicking on *Edit* when `LocalConnectionCLIENT` is selected will show an error.

5.1.2 Guide (core)

This section contains the guide to using the Discord Advertisement Framework at it's core (console) or in other words, running it directly by Python and providing all the required information inside a Python script (`.py` file).

Framework control (core)

The following section shows how DAF's core can be started / stopped. The first thing you need is the library installed, see *Installation*.

The framework can be started using `daf.core.run()` function and stopped with the `daf.core.shutdown()` function. It can also be stopped by a SIGINT signal, which can be signaled using CTRL+C. For full list of parameters refer to `daf.core.run()`'s description.

```python
import daf
daf.run()
```

The above code is somewhat useless as nothing was configured. *On the next page*, we will take a look on how to define our accounts, guilds (servers) and messages.

Shilling list definition (core)

This document holds information regarding shilling with message objects.

We will now see how our shilling / advertisement list can be defined.

Definition of accounts (core)

Discord accounts / clients are inside DAF represented by the *`daf.client.ACCOUNT`* class. It accepts many parameters, out of which these are the most important:

- `token`: A string (text) parameter. It is the token an account needs to login into Discord. Token can be obtained through the developer portal for bot accounts and through a browser for user accounts.
- `is_user`: An optional `True` / `False` parameter. Discord has 2 types of clients - user accounts and bots. Set this to True when the above `token` belongs to a user and not a bot.
- `proxy`: An optional string (text) parameter. Represents a proxy URL used to access Discord.
- `servers`: A list of servers. In DAF, servers are referred to as "guild", which was Discord's original name for a server. Elements inside this list can be any objects inherited from `daf.guild.BaseGUILD` class. Three types of servers exist (are inherited from `daf.guild.BaseGUILD`):
  - *`daf.guild.GUILD`*

    Represents an actual Discord server with channels and members.
  - *`daf.guild.USER`*

    Represents a user and their direct messages.

    Caution: Shilling to DM's is not recommended as there is no way to check if our client has permissions. There is a high risk of Discord automatically banning you if you attempt to shill messages to users, who can't receive them from you.
  - *`daf.guild.AutoGUILD`*

    Represents multiple Discord servers with channels and members, whose names match a configured pattern. Strictly speaking, this isn't actually inherited from `daf.guild.BaseGUILD`, but is rather a wrapper for multiple *`daf.guild.GUILD`*. It can be used to quickly define the entire the entire server list, without manually creating each *`daf.guild.GUILD`*.

    Refer to the *Automatic Generation (core)* section for more information.

Now let's see an example.

```python
from daf.client import ACCOUNT
import daf

accounts = [
```

(continues on next page)

(continued from previous page)

```python
    ACCOUNT(
        token="HHJSHDJKSHKDJASHKDASDHASJKDHAKSJDHSAJKHSDSAD",
        is_user=True,  # Above token is user account's token and not a bot token.
        servers=[]
    )
]

daf.run(accounts=accounts)
```

As you can see from the above example, the definition of accounts is rather simple. Notice we didn't define our servers. We will do that in the next section.

After running the example, the following output is displayed. Ignore the `intents` warnings for now. These warnings are not even relevant for user accounts. Intents are settings of what kind of events the *ACCOUNT* should listen to and are controlled with its `intents` parameter. User accounts have no notion of intents.

Notice the first line of the output. It tells us that the logs will be stored into a specific folder. DAF supports message logging, meaning that a message log is created for each sent message. A logger can be given to the `daf.core.run()`'s `logger` parameter. For more information about logging see *Logging (core)*.

```
[2024-01-21 13:24:22.887679] (NORMAL) | daf.logging.logger_file: LoggerJSON logs will be␣
↪saved to C:\Users\david\daf\History (None)
[2024-01-21 13:24:22.887679] (WARNING) | daf.client: Members intent is disabled, it is␣
↪needed for automatic responders' constraints and invite link tracking. (None)
[2024-01-21 13:24:22.887679] (WARNING) | daf.client: Message content intent is disabled,␣
↪it is needed for automatic responders. (None)
[2024-01-21 13:24:22.887679] (NORMAL) | daf.client: Logging in... (None)
[2024-01-21 13:24:25.910163] (NORMAL) | daf.client: Logged in as Aproksimacka (None)
[2024-01-21 13:24:25.910163] (NORMAL) | daf.core: Initialization complete. (None)
```

Definition of servers / guilds (core)

We will only cover the definition of `daf.guild.GUILD` here. We will not cover `daf.guild.USER` separately as the definition process is exactly the same. We will also not cover `daf.guild.AutoGUILD` here, as it is covered in *Automatic Generation (core)*.

Let's define our `daf.guild.GUILD` object now. Its most important parameters are:

- `snowflake`: An integer parameter. Represents a unique identifier, which identifies every Discord resource. Snowflake can be obtained by enabling the developer mode, right-clicking on the guild of interest, and then left-clicking on *Copy Server ID*.
- `messages`: A list parameter of our message objects. Message objects represent the content that will be sent into specific channels, with a specific period. For our `daf.guild.GUILD`, messages can be the following classes:
  - `daf.message.TextMESSAGE`: Message type for sending textual data. Data includes files as well.
  - `daf.message.VoiceMESSAGE`: Message type for sending audio data / playing audio to voice channels.

Let's expand our example from *Definition of accounts (core)*.

```python
from daf.client import ACCOUNT
from daf.guild import GUILD
import daf

```

(continues on next page)

(continued from previous page)

```python
accounts = [
    ACCOUNT(
        token="HHJSHDJKSHKDJASHKDASDHASJKDHAKSJDHSAJKHSDSAD",
        is_user=False,  # Above token is user account's token and not a bot token.
        servers=[
            GUILD(
                snowflake=863071397207212052,
                messages=[]
            )
        ]
    )
]

daf.run(accounts=accounts)
```

Now let's define our messages.

Definition of messages (core)

Three kinds of messages exist. Additional to *daf.message.TextMESSAGE* and *daf.message.VoiceMESSAGE*, is the *daf.message.DirectMESSAGE* message type. This message type is used together with *daf.guild.USER* for sending messages into DMs. Unlike the previously mentioned message types, *DirectMESSAGE* does not have the `channels` parameter.

Now let's describe some parameters. The most important parameters inside *daf.message.TextMESSAGE* are:

- `data`: A *TextMessageData* object or a *DynamicMessageData* (Dynamically obtained data) inherited object. It represents the data that will be sent into our text channels.
- `channels`: A list of integers or a single *AutoCHANNEL* object. The integers inside a list represents channel snowflake IDs. Obtaining the IDs is the same as for *guilds*. See *Automatic Generation (core)* for information about *AutoCHANNEL*.
- `period`: It represents the time period at which messages will be periodically sent. It can be one of the following types:
  - *FixedDurationPeriod*: A fixed time period.
  - *RandomizedDurationPeriod*: A randomized (within a certain range) time period.
  - *DaysOfWeekPeriod*: A period that sends at multiple specified days at a specific time.
  - *DailyPeriod*: A period that sends every day at a specific time.

Now that we have an overview of the most important parameters, let's define our message. We will define a message that sends fixed data into a single channel, with a fixed time (duration) period.

```python
from daf.message.messageperiod import FixedDurationPeriod
from daf.messagedata import TextMessageData
from daf.message import TextMESSAGE
from daf.client import ACCOUNT
from daf.guild import GUILD

from datetime import timedelta

import daf
```

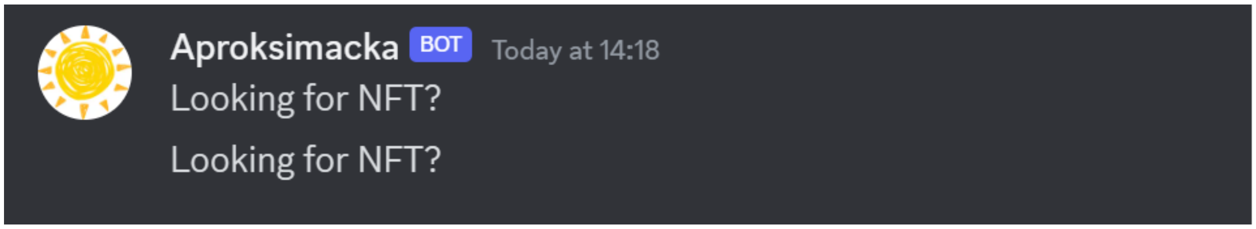
(continues on next page)

(continued from previous page)

```python

accounts = [
    ACCOUNT(
        token="HHJSHDJKSHKDJASHKDASDHASJKDHAKSJDHSAJKHSDSAD",
        is_user=False,  # Above token is user account's token and not a bot token.
        servers=[
            GUILD(
                snowflake=863071397207212052,
                messages=[
                    TextMESSAGE(
                        data=TextMessageData(content="Looking for NFT?"),
                        channels=[1159224699830677685],
                        period=FixedDurationPeriod(duration=timedelta(seconds=15))
                    )
                ]
            )
        ]
    )
]

daf.run(accounts=accounts)
```



Aproksimacka BOT Today at 14:18

Looking for NFT?

Looking for NFT?

Similarly to text messages, voice messages can be defined with *daf.message.VoiceMESSAGE*. Definition is very similar to *daf.message.TextMESSAGE*. The only thing that differs from the above example is the `data` parameter. That parameter is with *VoiceMESSAGE* of type *VoiceMessageData* (Fixed data) or *DynamicMessageData* (Dynamically obtained data). Additionally, it contains a `volume` parameter.

Message advertisement examples

The following examples show a complete core script setup needed to advertise periodic messages.

TextMESSAGE

Listing 5.2: TextMESSAGE full example

```python
"""
Example shows the basic message shilling into a fixed guild.
A text message is sent every 8-12 hours (randomly chosen) and it contains a text content␣
↪and 3 files.
The message will be first sent in 4 hours.
"""

```

(continues on next page)

(continued from previous page)

```python
# Import the necessary items
from daf.logging.logger_json import LoggerJSON

from daf.messagedata import FILE
from daf.messagedata.textdata import TextMessageData
from datetime import timedelta
from daf.client import ACCOUNT
from daf.message.text_based import TextMESSAGE
from daf.message.messageperiod import RandomizedDurationPeriod
from daf.guild.guilduser import GUILD
from daf.logging.tracing import TraceLEVELS
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\david\\daf\\History",
)


# Defined accounts
accounts = [
    ACCOUNT(
        token="TOKEN",
        is_user=True,
        servers=[
            GUILD(
                snowflake=123456789,
                messages=[
                    TextMESSAGE(
                        data=TextMessageData(
                            content="Buy our red dragon NFTs today!",
                            files=[
                                FILE(filename="C:/Users/david/Downloads/Picture1.png"),
                                FILE(filename="H:/My Drive/PR/okroznice/2023/2023_01_28 -
↪ Skiing.md"),
                                FILE(filename="H:/My Drive/PR/okroznice/2023/2023_10_16 -
↪ Volitve SSFE.md"),
                            ],
                        ),
                        channels=[123123231232131231, 329312381290381208321,␣
↪3232320381208321],
                        period=RandomizedDurationPeriod(
                            minimum=timedelta(hours=8.0),
                            maximum=timedelta(hours=12.0),
                            next_send_time=timedelta(hours=4.0),
                        ),
                    ),
                ],
                logging=True,
            ),
        ],
    ),
```

(continues on next page)

(continued from previous page)

```python
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL,
)
```

VoiceMESSAGE

Listing 5.3: VoiceMESSAGE full example

```python
"""
Example shows the basic message shilling into a fixed guild.
A text message is sent every 30-90 minutes (randomly chosen) and it plays a single audio␣
↪file.
The message will be first sent in 10 minutes.
"""

# Import the necessary items
from daf.logging.logger_json import LoggerJSON

from daf.messagedata import FILE
from daf.messagedata.voicedata import VoiceMessageData
from datetime import timedelta
from daf.client import ACCOUNT
from daf.message.voice_based import VoiceMESSAGE
from daf.message.messageperiod import RandomizedDurationPeriod
from daf.guild.guilduser import GUILD
from daf.logging.tracing import TraceLEVELS
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\david\\daf\\History",
)

# Defined accounts
accounts = [
    ACCOUNT(
        token="TOKEN",
        is_user=True,
        servers=[
            GUILD(
                snowflake=318732816317823168276278,
                messages=[
                    VoiceMESSAGE(
                        data=VoiceMessageData(FILE(filename="./VoiceMESSAGE.mp3")),
                        channels=[1136787403588255784, 1199125280149733449],
```

(continues on next page)

(continued from previous page)

```python
                        period=RandomizedDurationPeriod(
                            minimum=timedelta(minutes=30),
                            maximum=timedelta(minutes=90),
                            next_send_time=timedelta(minutes=10),
                        ),
                    ),
                ],
                logging=True,
            ),
        ],
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL,
)
```

DirectMESSAGE

Listing 5.4: DirectMESSAGE full example

```python

"""
Example shows the basic message shilling into a fixed DM.
A text message is sent every 8-12 hours (randomly chosen) and it contains a text content␣
↪and 3 files.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
WARNING! Using this do directly shill to DM is VERY DANGEROUS!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
There is no way to check for permissions with DM messages, thus the client
may try to make many forbidden requests. This will eventually result in Discord
banning. USE AT YOUR OWN RISK!!!!

If you just want to respond to DM messages, the DMResponder is a better example.
"""

# Import the necessary items
from daf.logging.logger_json import LoggerJSON

from daf.messagedata import FILE
from daf.messagedata.textdata import TextMessageData
from datetime import timedelta
from daf.client import ACCOUNT
from daf.message.text_based import DirectMESSAGE
from daf.message.messageperiod import RandomizedDurationPeriod
from daf.guild.guilduser import USER
from daf.logging.tracing import TraceLEVELS
```

(continues on next page)

(continued from previous page)

```python
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\david\\daf\\History",
)


# Defined accounts
accounts = [
    ACCOUNT(
        token="TOKEN",
        is_user=False,
        servers=[
            USER(
                snowflake=145196308985020416,
                messages=[
                    DirectMESSAGE(
                        data=TextMessageData(
                            content="Buy our red dragon NFTs today!",
                            files=[
                                FILE(filename="C:/Users/david/Downloads/Picture1.png"),
                                FILE(filename="H:/My Drive/PR/okroznice/2023/2023_01_28 -
↪ Skiing.md"),
                                FILE(filename="H:/My Drive/PR/okroznice/2023/2023_10_16 -
↪ Volitve SSFE.md"),
                            ],
                        ),
                        period=RandomizedDurationPeriod(
                            minimum=timedelta(hours=8.0),
                            maximum=timedelta(hours=12.0),
                        ),
                    ),
                ],
                logging=True,
            ),
        ],
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL,
)
```

Next up, we will take a look how to setup and use *message logging*.

Logging (core)

Changed in version v2.7: Added **invite link tracking**

The logging module is responsible for 2 types of logging:

1. **Messages** - Logs (attempts of) sent messages
2. **Invite links** - Tracks new member joins with configured invite links.

Warning: To track invite links, the Members intent (event setting) is needed. To use invite link tracking, users need to enable the privileged intent 'SERVER MEMBERS INTENT' inside the Discord developer portal and also set the `members` intent to True inside the `intents` parameter of *`ACCOUNT`*.

Invites intent is also needed. Enable it by setting `invites` to True inside the `intents` parameter of *`ACCOUNT`*.

Invite link tracking is **bot account** only and does not work on user accounts.

Logging can be enabled for each *`GUILD`* / *`USER`* / *`AutoGUILD`* if the `logging` parameter is set to `True`.

Note: **Invite links** will be tracked regardless of the GUILD's `logging` parameter. Invite link tracking is configured solely by the `invite_track` parameter inside *`GUILD`*.

Logging is handled thru so called **logging managers** and currently 3 exist:

- LoggerJSON: (default) Logging into JSON files. (*JSON Logging (file)*)
- LoggerSQL: Logging into a relational database (local or remote). (*Relational Database Log (SQL)*)
- LoggerCSV: Logging into CSV files, where certain fields are still JSON. (*CSV Logging (file)*)

Each logging manager can have a backup manager specified by it's `fallback` parameter. If current manager fails, it's fallback manager will be used temporarily to store the log. It will only use the fallback once and then, at the next logging attempt, the original manager will be used.

JSON Logging (file)

The logs are written in the JSON format and saved into a JSON file, that has the name of the guild / user you were sending messages into. The JSON files are grouped 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
```

Logging layer

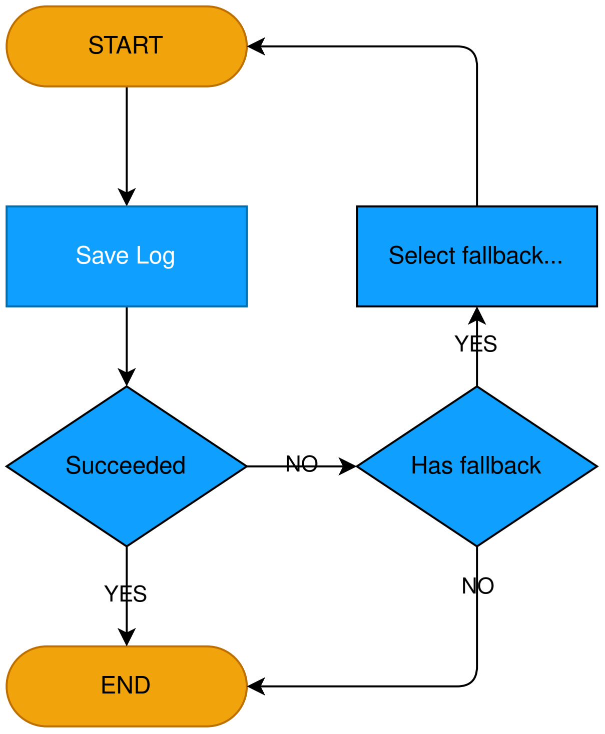


Fig. 5.4: Logging process with fallback

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.
- "invite_tracking": Dictionary that holds invite link tracking information.

  It's keys are invite link ID's (final part of invite link URL) and the value is a list of invite link logs, where a new log is created on each member join.

  Each invite log is a dictionary and contains the following keys:

  - "id": Member's snowflake (Discord) ID,
  - "name": Member's username,
  - "index": serial number of the log,
  - "timestamp": Date-Time when the log was created.
- "message_tracking": Dictionary that holds information about sent messages.

  Note: Only messages sent from DAF are tracked. Other messages are not tracked.

  The keys are snowflake IDs of each each account who has sent the message from DAF.

  The value under each key is a dictionary containing:

  - "name": Name of the sender (author)
  - "id": Snowflake ID of the sender
  - "messages": List of previously sent messages by the corresponding author with their context. It is message type dependent and is generated in:
    * *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
```

(continues on next page)

(continued from previous page)

```
│       └──13
|           └── #David's dungeon.csv
```

CSV structure

Warning: Invite link tracking is not supported with CSV logging.

The structure contains the following attributes:

- Index (integer) - this is a unique ID,
- Timestamp (string)
- Guild Type (string),
- Guild Name (string),
- Guild Snowflake (integer),
- Author name (string),
- Author 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)

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

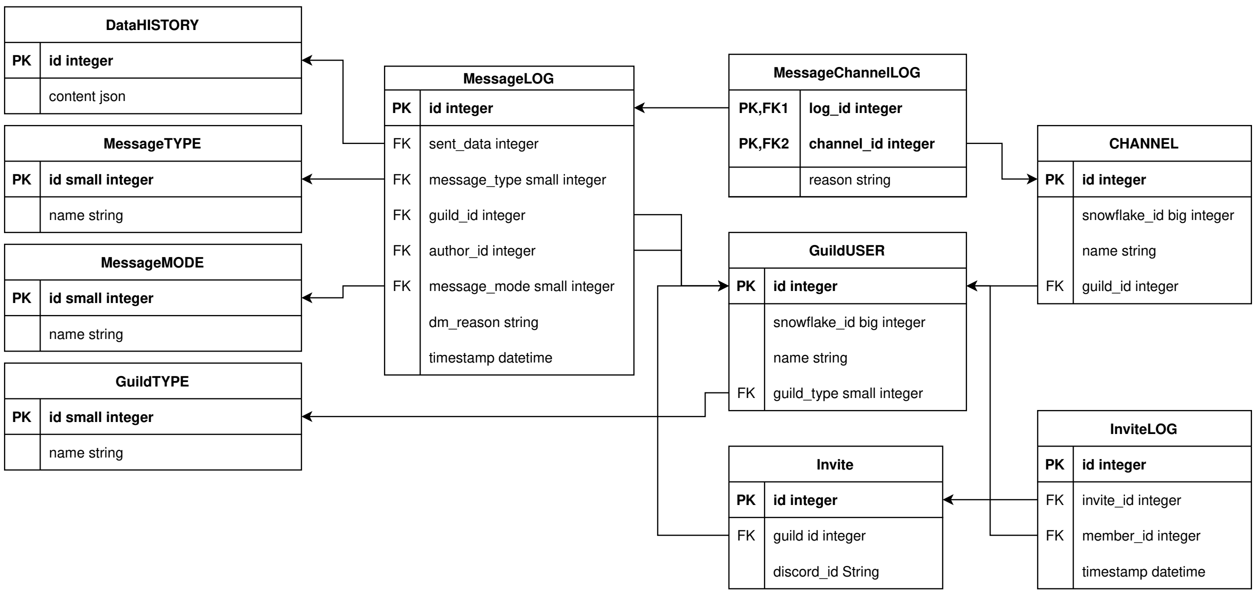
For daf to use SQL logging, you need to pass the *run()* function with the `logger` parameter and pass it the *LoggerSQL* object.

Features

- Multiple dialects (sqlite, mssql, postgresql, mysql)
- Automatic creation of the schema
- Caching for faster logging
- Low redundancy for reduced file size
- Automatic error recovery

Warning: The database must already exist (unless using SQLite). However it can be completely empty, no need to manually create the schema.

ER diagram



Analysis

The `LoggerSQL` provides some methods for data analysis:

- For message history:
  - `analytic_get_num_messages()`
  - `analytic_get_message_log()`
- For invite link tracking:
  - `analytic_get_num_invites()`
  - `analytic_get_invite_log()`

SQL 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, represents guild id of guild the message was sent into.
- author_id: Integer - Foreign key pointing to *GuildUSER* table, represents the author account of the message.
- 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 and all the authors.

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.

Invite

Description

Table that represents tracked invite links.

Attributes

- **[Primary Key]** id: Integer - Internal ID of the invite inside the database.
- **[Foreign Key]** guild_id: Integer - Foreign key pointing to a row inside the *GuildUSER* table (The guild that owns the invite).
- discord_id: String - Discord’s invite ID (final part of the invite URL).

InviteLOG

Description

Table which’s entries are logs of member joins into a guild using a specific invite link.

Attributes

- **[Primary Key]** id: Integer - Internal ID of the log inside the database.
- **[Foreign Key]** invite_id: Integer - Foreign key pointing to a row inside the *Invite* table. Describes the link member used to join a guild.

- **[Foreign Key]** member_id: Integer - Foreign key pointing to a row inside the *GuildUSER* table. Describes the member who joined.
- timestamp: DateTime - The date and time a member joined into a guild.

Matching logic

Before going into the automatic guild definition and automatic channel definition, we first need to explain how their patterns are matched. Automatic guild definition, automatic channel definition and also *Automatic responder* determine the match by considering a logic pattern. The logic pattern can either be an actual logical operations (and, or, not) or a text-matching operation.

Text matching operations

These operations are used for matching actual text. Two types currently exist. They are the RegEx type, for matching based on a regex pattern, and a contains type, for matching when the text input contains the (single) configured keyword.

> **Caution:** Text matching is **lower-case**. This means that all the message content will be interpreted as lower-case characters.
>
> When writing patterns, make sure the pattern words are lower-case.

`regex`

This can be used for matching a RegEx (regular expression pattern). It is probably the most desirable as it allows flexible pattern matching. For example, if we want the message to contain the words "buy" and "nft", we can write our pattern like this: `buy.*nft`. The former pattern will match when the message contains both words - "buy" and "nft", and "nft" appears after "buy". The `.*` means match any character (`.`) zero or infinite times (`*`).

Here are some example messages the `buy.*nft` pattern will match:

- "Where can I buy the NFTs?"
- "May I buy the dragon NFT from you?"
- "I would really want to buy the bunny NFT."

Additionally, RegEx supports the logical *OR* operation. This can be done by separating multiple RegEx patterns with the `|` character. For example, the `buy.*nft|sell.*nft` pattern will match the text message if any of the 2 patterns (separated by `|`) matches.

> **Caution:** RegEx is sensitive to spaces around the `|` (the logical OR) character. When a space is inserted into a RegEx pattern, it is considered as something that needs to appear for the text to be matched.
>
> For example, pattern `hello | world` would match if the text contains the "hello " sequence or the " world" sequence (notice the spaces).
>
> Thus, unless you want spaces to be matched as well, write `buy.*nft|sell.*nft` instead of `buy.*nft | sell.*nft`! This is especially important for server (guild) names.

For testing RegEx patterns, the following site is recommended: https://regex101.com/.

contains

This can be used for matching text messages containing a certain a word. As a parameter it accepts the word (`keyword`) used for checking. Usually, *`contains`* would be used alongside logical operations, such as *`or_`*, to match any of the multiple words.

Logical operations

Logical operations are used to combine multiple text matching operations, as well as other nested logical operations. Themselves, they do not match the text inside a message.

and_

Represents a logical *AND* operation. *`and_`* evaluates to true when all of the operands inside evaluate to true.

For example, if we write:

```
and_(contains('buy'), contains('nft'), contains('dragon'))
```

then the text message will be matched only if it contains all of the words "buy", "nft" and "dragon" (in any order). The above example would in a human-readable form look like `contains('buy') and contains('nft') and contains('dragon')`, where the `contains('word')` evaluates to a human-readable form of `if 'word' is in message`.

or_

Represents a logical *OR* operation. *`or_`* evaluates to true when any of the operands inside evaluate to true.

For example, if we write:

```
or_(contains('buy'), contains('nft'), contains('dragon'))
```

then the text message will be matched only if it contains any of the words "buy", "nft" and "dragon" (in any order). The above example would in a human-readable form look like `contains('buy') or contains('nft') or contains('dragon')`.

not_

Represents a logical *NOT* operation. *`not_`* accepts a single operand and evaluates to true when that operand is false. Basically, it negates the operand.

For example, if we write:

```
and_(contains('buy'), not_(contains('dragon')))
```

then the text message will be matched only if it contains the word "buy" but doesn't contain the word "dragon". The above example would in a human-readable form look like `contains('buy') and not contains('dragon')`.

Automatic Generation (core)

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

Shilling scheme generation

While the servers can be manually defined and configured, which can be time consuming if you have a lot of guilds to shill into, DAF also supports automatic definition of servers and channels. Servers and channels can be automatically defined by creating some matching rules described in the *Matching logic* chapter of this guide.

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 automatically managed servers, use *AutoGUILD* which internally generates *GUILD* instances. Simply create a list of *AutoGUILD* objects and then pass it to the `servers` parameter of *daf.client.ACCOUNT*.

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.

```python
# Import the necessary items
from daf.logging.logger_json import LoggerJSON

from daf.client import ACCOUNT
from daf.logic import contains
from daf.guild.autoguild import AutoGUILD
from daf.logic import or_
from daf.logging.tracing import TraceLEVELS
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\david\\daf\\History",
)

# Define remote control context


# Defined accounts
accounts = [
    ACCOUNT(
        token="TOKEN_HERE",
        is_user=True,
        servers=[
            AutoGUILD(
                include_pattern=or_(
                    operands=[
```

(continues on next page)

(continued from previous page)

```
                        contains(keyword="shill"),
                        contains(keyword="NFT"),
                        contains(keyword="dragon"),
                        contains(keyword="promo"),
                    ],
                ),
                logging=True,
            ),
        ],
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL,
    save_to_file=False
)
```

Automatic channel generation

For a automatically managed channels, use *AutoCHANNEL*. It can be passed to xMESSAGE objects to the `channels` parameters.

Listing 5.5: AutoCHANNEL example

```
# Import the necessary items
from daf.logging.logger_json import LoggerJSON

from daf.guild.autoguild import AutoGUILD
from daf.logic import or_
from daf.messagedata.textdata import TextMessageData
from datetime import timedelta
from daf.message.autochannel import AutoCHANNEL
from daf.message.messageperiod import FixedDurationPeriod
from daf.message.text_based import TextMESSAGE
from daf.logic import contains
from daf.client import ACCOUNT
from daf.logging.tracing import TraceLEVELS
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\david\\daf\\History",
)

# Define remote control context


# Defined accounts
```

(continues on next page)

(continued from previous page)

```
accounts = [
    ACCOUNT(
        token="TOKEN_HERE",
        is_user=True,
        servers=[
            AutoGUILD(
                include_pattern=or_(
                    operands=[
                        contains(keyword="shill"),
                        contains(keyword="NFT"),
                        contains(keyword="dragon"),
                        contains(keyword="promo"),
                    ],
                ),
                messages=[
                    TextMESSAGE(
                        data=TextMessageData(content="Checkout my  new Red Dragon NFT!␣
↪Additionaly, we also have the Golden Dragon - limited time only!"),
                        channels=AutoCHANNEL(
                            include_pattern=or_(
                                operands=[
                                    contains(keyword="nft"),
                                    contains(keyword="shill"),
                                    contains(keyword="self-promo"),
                                    contains(keyword="projects"),
                                    contains(keyword="marketing"),
                                ],
                            ),
                        ),
                        period=FixedDurationPeriod(duration=timedelta(seconds=5.0)),
                    ),
                ],
                logging=True,
            ),
        ],
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL,
    save_to_file=False
)
```

Automatic responder

New in version 4.0.0.

Discord Advertisement Framework also supports the so called automatic responder.

The automated responder can promptly reply to messages directed to either the guild or private messages. It can be customized to trigger based on specific keyword patterns within a message, provided the constrains are met.

Warning: For the automatic responder to work, *ACCOUNT*'s `intents` parameter needs to have the following intents enabled:

- members
- guild_messages
- dm_messages
- message_content

```
intents = Intents.default()
intents.members=True
intents.guild_messages=True
intents.dm_messages=True
intents.message_content=True
```

There are 2 responder classes:

- *daf.responder.DMResponder* - responds to messages sent into the bot's direct messages.
- *daf.responder.GuildResponder* - responds to messages sent into a guild channel. The bot must be joined into that guild, otherwise it will not see the message.

They can be used on each account through the *ACCOUNT*'s `responders` parameter. The parameter is a list of responders.

```
import daf

daf.client.ACCOUNT(
  ...,
  responders=[daf.responder.DMResponder(...), ...]
)
```

Both responders accept the following parameters:

condition
Represents the message match condition. If both the condition and the constraints are met, then an action (response) will be triggered.

The *Matching logic* chapter explains how matching is done (same as *daf.guild.AutoGUILD* and *daf.message.AutoCHANNEL*)

action
Represent the action taken upon message match (and constraint fulfillment). Currently the only action is a message response. There are 2 types of responses, which both send a message in response to the original trigger message.

DMResponse Will send a reply to the message author's private messages.

GuildResponse Will send a reply to same channel as the trigger message.

This is only available for the *GuildResponder* responder.

Both response classes accept a single parameter `data` of base type `BaseTextData`, which represents the data that will be sent in the response message. The `BaseTextData` type has two different implementations:

- *TextMessageData* - for fixed data

```
import daf

daf.responder.DMResponse(
    data=daf.messagedata.TextMessageData("My content")
)
```

- *DynamicMessageData* - for data obtained through a function.

```
import daf
import requests

class MyCustomText(DynamicMessageData):
    def __init__(self, a: int):
        self.a = a

    def get_data(self):  # Can also be async
        mydata = requests.get('https://daf.davidhozic.com').text
        return TextMessageData(content=mydata)

daf.responder.DMResponse(
    data=MyCustomText(5)
)
```

constraints

In addition to the `condition` parameter, constrains represent extra checks. These checks (constraints) must all be fulfilled, otherwise no reply will me made to the trigger message.

A constraint can be any class implementing a constraint interface. The constraint interface is different for different responder types:

daf.responder.DMResponder Constraints are implementations of the *BaseDMConstraint* interface.

Built-in implementations:

- *MemberOfGuildConstraint*

daf.responder.GuildResponder Constraints are implementations of the *BaseGuildConstraint* interface.

Built-in implementations:

- *GuildConstraint*

In addition to the built-in implementations, custom constrains can be made by implementing one of the two interfaces.

Listing 5.6: Custom constraint implementation

```
from daf import discord
from daf.responder import GuildResponder
```

(continues on next page)

(continued from previous page)

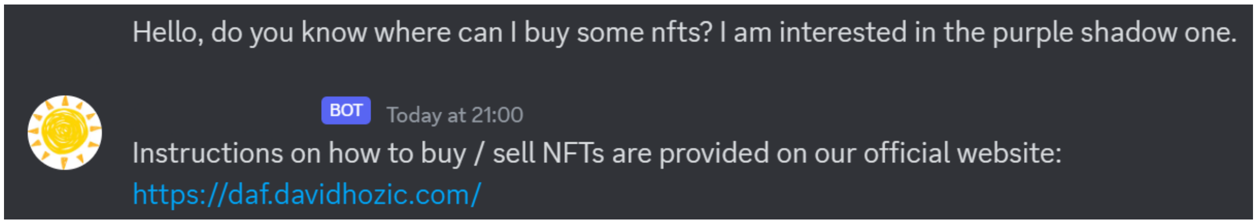
```
from daf.responder.constraints import BaseGuildConstraint

class MyConstraint(BaseGuildConstraint):
    def __init__(self, <some parameters>): ...

    def check(self, message: discord.Message, client: discord.Client) ->␣
↪bool:
        return <True / False>

GuildResponder(..., constraints=[MyConstraint(...)])
```

Here is a full example of a DM responder:


Hello, do you know where can I buy some nfts? I am interested in the purple shadow one.

BOT Today at 21:00

Instructions on how to buy / sell NFTs are provided on our official website: https://daf.davidhozic.com/

```
"""
The example shows how to use the automatic responder feature.

DMResponder is used. It listens to messages inside DMs.
RegEx (regex) is used to match the text pattern.
MemberOfGuildConstraint is used for ensuring the author of the dm message is a member
of guild with ID 863071397207212052.
"""
# Import the necessary items
from _discord.flags import Intents
from daf.messagedata.textdata import TextMessageData
from daf.responder.constraints.dmconstraint import MemberOfGuildConstraint
from daf.client import ACCOUNT
from daf.responder.dmresponder import DMResponder
from daf.responder.actions.response import DMResponse
from daf.logic import regex
import daf


# Defined accounts
intents = Intents.default()
intents.members=True
intents.guild_messages=True
intents.dm_messages=True
intents.message_content=True

accounts = [
    ACCOUNT(
        token="CLIENT_TOKEN_HERE",
        is_user=False,
        intents=intents,
```

(continues on next page)

(continued from previous page)

```
        responders=[
            DMResponder(
                condition=regex(
                    pattern="(buy|sell).*nft",
                ),
                action=DMResponse(
                    data=TextMessageData(
                        content="Instructions on how to buy / sell NFTs are provided on␣
↪our official website:\nhttps://daf.davidhozic.com",
                    ),
                ),
                constraints=[
                    MemberOfGuildConstraint(
                        guild=863071397207212052,
                    ),
                ],
            ),
        ],
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts
)
```

In the above example, we can see that the *daf.responder.DMResponder* has a `regex` condition defined. The RegEx pattern is as follows: `(buy|sell).*nft`. We can interpret the pattern as "match the message when it contains either the word "buy" or "sell", followed by any text as long as "nft" also appears (after "buy" / "sell") in that text. We can also see a *MemberOfGuildConstraint* instance. It is given a `guild=863071397207212052` ID parameter, which only allows a response if the author of the DM message is a member of the guild (server) with ID 863071397207212052.

The `intents` parameter of our *ACCOUNT* defines 4 intents. Intents are settings of which events Discord will send to our client. Without them the auto responder does not work.

Web browser (core)

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.

The web browser functionality includes the login (via browser) with email and password as well as the automatic guild join (also via browser).

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.

```python
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.

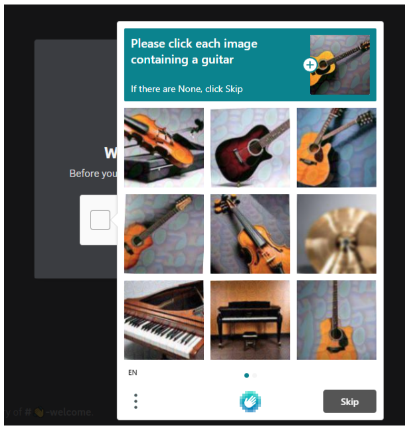
If you restart DAF, it will not re-login, but will just load the data from the saved storage under "daf_web_data" folder.

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. DAF will join a new guild every 45 seconds until the limit is reached.

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.


Please click each image containing a guitar
If there are None, click Skip
Before you
EN
Skip

```
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.

```
"""
Example shows login with email and password.
The AutoGUILD defines an auto_join parameter, which will be used to (semi) automatically
join new guilds based on a prompt. You as a user still need to solve any CAPTCHA␣
↪challenges.
"""

# Import the necessary items
from daf.web import QueryMembers
from daf.client import ACCOUNT
from daf.web import GuildDISCOVERY
from daf.web import QuerySortBy
from daf.guild.autoguild import AutoGUILD
from daf.logging.tracing import TraceLEVELS
import daf


# Defined accounts
accounts = [
    ACCOUNT(
        servers=[
            AutoGUILD(
                include_pattern=".*",
                auto_join=GuildDISCOVERY(
                    prompt="NFT arts",
                    sort_by=QuerySortBy.TOP,
                    total_members=QueryMembers.B100_1k,
                    limit=30,
                ),
            ),
        ],
        username="username@email.com",
        password="password",
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    debug=TraceLEVELS.NORMAL,
)
```

Dynamic modification (core)

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.

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

```python
"""
The example shows how to dynamically add objects to the framework
after it had already been run.
"""
from datetime import timedelta
```

(continues on next page)

(continued from previous page)

```python
import daf


async def user_task():
    # Returns the client to send commands to discord, for more info about client see␣
↪https://docs.pycord.dev/en/master/api.html?highlight=discord%20client#discord.Client
    account = daf.ACCOUNT(token="ASDASJDAKDK", is_user=False)
    guild = daf.GUILD(snowflake=123456)
    data_to_shill = daf.TextMessageData(
        "Hello World",
        daf.discord.Embed(title="Example Embed", color=daf.discord.Color.blue(),␣
↪description="This is a test embed")
    )
    text_msg = daf.TextMESSAGE(
        data=data_to_shill,
        channels=[12345, 6789],
        period=daf.FixedDurationPeriod(timedelta(minutes=1))
    )

    # Dynamically add account
    await daf.add_object(account)

    # Dynamically add guild
    await account.add_server(guild)
    # await daf.add_object(guild, account)

    # Dynamically add message
    await guild.add_message(text_msg)
    # await daf.add_object(text_msg, guild)


#########################################################################################
↪###
if __name__ == "__main__":
    daf.run(user_callback=user_task)
```

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()`.

```python
"""
The following example uses a predefined list of messages to shill.
When the user task is run, it removes the message from the shill list dynamically.
"""
import daf

accounts = [
    account := daf.ACCOUNT(
        token="SDSADSDA87sd87",
        is_user=False,
```

(continues on next page)

(continued from previous page)

```
        servers=[
            daf.GUILD(snowflake=213323123, messages=[]) # No messages as not needed for␣
↪this demonstration
        ]
    )
]


async def user_task():
    guild = account.servers[0]
    await daf.remove_object(guild)


#########################################################################################
↪###
if __name__ == "__main__":
    daf.run(user_callback=user_task, accounts=accounts)

```

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:

```
"""
The following example shows how to update the object to run on
new parameters.
It is generally not recommended to use this if there is a better way,
for example if you want to change the message, it's better to just create
a new message object, however the only way to update SQL manager for logging is via the
update method.
"""
from datetime import timedelta
import asyncio
import daf


message = daf.TextMESSAGE(
    data=daf.TextMessageData("Hello"),
    channels=[123455, 1425215],
    period=daf.FixedDurationPeriod(timedelta(seconds=5))
)
```

(continues on next page)

(continued from previous page)

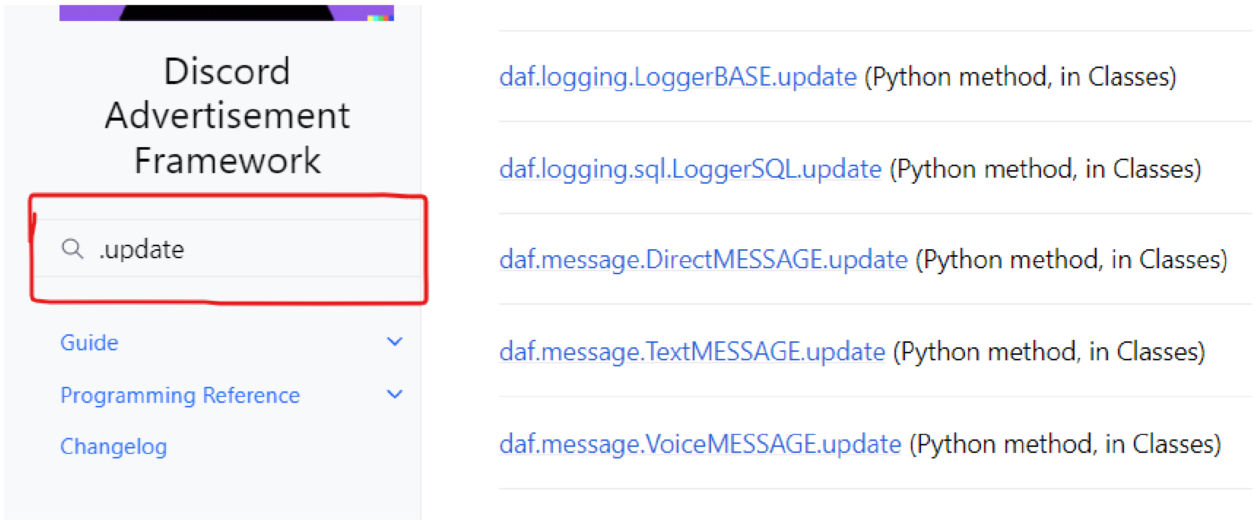
```python

accounts = [
    daf.ACCOUNT(
        token="SKJDSKAJDKSJDKLAJSKJKJKGSAKGJLKSJG",
        is_user=False,
        servers=[
            daf.GUILD(1234567, [message])
        ]
    )
]


async def user_task():
    # Will send "Hello" every 5 seconds
    await asyncio.sleep(10)
    await message.update(
        period=daf.FixedDurationPeriod(timedelta(seconds=20)),
    )
    # Will send "Hello" every 20 seconds

if __name__ == "__main__":
    daf.run(user_callback=user_task, accounts=accounts)
```

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)

Remote control (core)

New in version 2.8.

While DAF can run completely standalone locally, it also allows to be run as a server that will accept connections from a graphical interface (*Remote control (GUI)*).

The remote module spins up a HTTP server, which can also be given a certificate and a private key allowing HTTPS connections.

To set up the core as a remote server, pass the *run()* function with the `remote_client` parameter. It accepts an object of type *daf.remote.RemoteAccessCLIENT*.

After the script is ran, DAF will listen and accept connections based on the configured options. While the server is running, DAF can be used the same way as if there was no server at all.

5.2 API reference

5.2.1 Program reference

Contain classes and functions description of the Program API.

Auto responder

ConstraintBase

class `daf.responder.constraints.baseconstraint.`**`ConstraintBase`**

abstract check(*message: Message*, *client: Client*) → bool

Verifies if the constraint is fulfilled.

Parameters

message (`discord.Message`) – Potential message to be responded to.

BaseGuildConstraint

class `daf.responder.constraints.guildconstraint.`**`BaseGuildConstraint`**

Constraint base for all guild constraints.

abstract check(*message: Message*, *client: Client*) → bool

Verifies if the constraint is fulfilled.

Parameters

message (`discord.Message`) – Potential message to be responded to.

GuildConstraint

class `daf.responder.constraints.guildconstraint.`**`GuildConstraint`**(*guild: int | Guild*)

Constraint that checks if the message originated from the specific `guild`.

Parameters

guild (*int | discord.Guild*) – The guild to which the message must be sent for the constraint to be fulfilled.

check(*message: Message*, *client: Client*) → bool

Verifies if the constraint is fulfilled.

Parameters

message (`discord.Message`) – Potential message to be responded to.

BaseDMConstraint

class `daf.responder.constraints.dmconstraint.`**`BaseDMConstraint`**

Base for constraints that are DM specific.

abstract check(*message: Message*, *client: Client*) → bool

Verifies if the constraint is fulfilled.

Parameters

message (`discord.Message`) – Potential message to be responded to.

MemberOfGuildConstraint

class `daf.responder.constraints.dmconstraint.`**`MemberOfGuildConstraint`**(*guild: int | Guild*)

Constraint that checks if the DM message author is part of the `guild`.

Parameters

guild (*`int | discord.Guild`*) – The guild to which the message's author must belong for the constraint to be fulfilled.

check(*message: Message*, *client: Client*) → bool

Verifies if the constraint is fulfilled.

Parameters

message (`discord.Message`) – Potential message to be responded to.

BaseResponse

class `daf.responder.actions.response.`**`BaseResponse`**(*data: BaseTextData*)

Base response class.

Parameters

data (*`BaseTextData`*) – The data that can be sent into the text / DM channel.

abstract async perform(*message: Message*)

Perform the action.

Parameters

message (*`discord.Message`*) –

DMResponse

class `daf.responder.actions.response.`**`DMResponse`**(*data: BaseTextData*)

DM response class. Used for responding into DM messages of the message's author.

Parameters

data (*`BaseTextData`*) – The data that will be sent into message author's DM channel.

async perform(*message: Message*)

Perform the action.

Parameters

message (*`discord.Message`*) –

GuildResponse

`class daf.responder.actions.response.`**`GuildResponse`**`(`*data: BaseTextData*`)`

Guild response class. Used for responding into the same channel as the message that triggered the response. The response is a reply.

Parameters

`data` (*`BaseTextData`*) – The data that will be sent into the channel.

`async` **`perform`**`(`*message: Message*`)`

Perform the action.

Parameters

`message` (*`discord.Message`*) –

DMResponder

`class daf.responder.`**`DMResponder`**`(`*condition: BaseLogic*, *action:* DMResponse, *constraints: List*[BaseDMConstraint] *= []*`)`

DM responder implementation. The responder is an object capable of making automatic replies to messages based on some keyword condition and constraints.

Parameters

- **`condition`** (*`BaseLogic`*) – The match condition. The condition represents the match condition of message's text.
- **`action`** (`BaseResponse`) – Represents the action taken on both `condition` and `constraints` being fulfilled.
- **`constraints`** (*`list`*`[ConstraintBase]`) – In addition to `condition`, constraints add additional checks that need to be fulfilled before performing an action. All of the constraints inside the `constraints` list need to be fulfilled.

`async` **`handle_message`**`(`*message: Message*`)`

Processes message and performs an action if all constraints satisfied.

GuildResponder

`class daf.responder.`**`GuildResponder`**`(`*condition: BaseLogic*, *action:* BaseResponse, *constraints: List*[BaseGuildConstraint] *= []*`)`

Guild responder implementation. The responder is an object capable of making automatic replies to messages based on some keyword condition and constraints.

Parameters

- **`condition`** (*`BaseLogic`*) – The match condition. The condition represents the match condition of message's text.
- **`action`** (`BaseResponse`) – Represents the action taken on both `condition` and `constraints` being fulfilled.
- **`constraints`** (*`list`*`[ConstraintBase]`) – In addition to `condition`, constraints add additional checks that need to be fulfilled before performing an action. All of the constraints inside the `constraints` list need to be fulfilled.

async handle_message(*message: Message*)

Processes message and performs an action if all constraints satisfied.

Logging reference

TraceLEVELS

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

Levels of trace for debug.

See also:

trace

Changed in version v2.3: Added DEPRECATION

Valid values are as follows:

`DEPRECATED = <TraceLEVELS.DEPRECATED: 0>`

`ERROR = <TraceLEVELS.ERROR: 1>`

`WARNING = <TraceLEVELS.WARNING: 2>`

`NORMAL = <TraceLEVELS.NORMAL: 3>`

`DEBUG = <TraceLEVELS.DEBUG: 4>`

trace

`daf.logging.tracing.`**`trace`**(*message: str, level:* TraceLEVELS | *int = TraceLEVELS.NORMAL, reason: Exception | None = None, exception_cls: Exception | None = 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.
- **`exception_cls`** (*`Optional[Exception]`*) – An exception to raise after tracing.

get_logger

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

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

Return type
LoggerBASE

LoggerBASE

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

Changed in version v2.7: Invite link tracking.

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

`abstract async analytic_get_num_messages(`*guild: int | None = None, author: int | None = None, after: datetime | None = None, before: datetime | None = None, guild_type: Literal['USER', 'GUILD'] | None = None, message_type: Literal['TextMESSAGE', 'VoiceMESSAGE', 'DirectMESSAGE'] | None = None, sort_by: Literal['successful', 'failed', 'guild_snow', 'guild_name', 'author_snow', 'author_name'] = 'successful', sort_by_direction: Literal['asc', 'desc'] = 'desc', limit: int = 500, group_by: Literal['year', 'month', 'day'] = 'day'*`)` → List[Tuple[date, int, int, int, str, int, str]]

Counts all the messages in the configured group based on parameters.

Parameters

- **guild** (*int*) – The snowflake id of the guild.
- **author** (*int*) – The snowflake id of the author.
- **after** (*Union[datetime, None] = None*) – Only count messages sent after the datetime.
- **before** (*Union[datetime, None]*) – Only count messages sent before the datetime.
- **guild_type** (*Literal["USER", "GUILD"] | None,*) – Type of guild.
- **message_type** (*Literal["TextMESSAGE", "VoiceMESSAGE", "DirectMESSAGE"] | None,*) – Type of message.
- **sort_by** (*Literal["successful", "failed", "guild_snow", "guild_name", "author_snow", "author_name"],*) – Sort items by selected. Defaults to "successful"

- **`sort_by_direction`** (*`Literal["asc", "desc"]`*) – Sort items by `sort_by` in selected direction (asc = ascending, desc = descending). Defaults to "desc"
- **`limit`** (*`int = 500`*) – Limit of the rows to return. Defaults to 500.
- **`group_by`** (*`Literal["year", "month", "day"]`*) – Results returned are grouped by `group_by`

Returns

List of tuples.

Each tuple contains:

- Date
- Successful sends
- Failed sends
- Guild snowflake id,
- Guild name
- Author snowflake id,
- Author name

Return type

list[tuple[date, int, int, int, str, int, str]]

`abstract async delete_logs`(*table: Any*, *logs: List[Any]*)

Method used to delete log objects objects.

Parameters

- **`table`** (*`Any`*) – The logging table to delete from.
- **`primary_keys`** (*`List[int]`*) – List of Primary Key IDs that match the rows of the table to delete.

`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 = '/home/docs/daf/History'*, *fallback: LoggerBASE | None = None*)

Changed in version v3.1: The index of each log is now a snowflake ID. It consists of <timestamp in ms since epoch> | <sequence number>. <sequence number> is of fixed size 8 bits, while timestamp is not of fixed size.

Changed in version v2.8: When file reaches size of 100 kilobytes, a new file is created.

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** (*Optional[str]*) – Path to the folder where logs will be saved. Defaults to /<user-home>/daf/History
- **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 delete_logs`(*logs: List[dict]*)

Method used to delete log objects objects.

Parameters

- **table** (*Any*) – The logging table to delete from.
- **primary_keys** (*List[int]*) – List of Primary Key IDs that match the rows of the table to delete.

`async analytic_get_num_messages`(*guild: int | None = None, author: int | None = None, after: datetime = datetime.datetime(1, 1, 1, 0, 0), before: datetime = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999), guild_type: Literal['USER', 'GUILD'] | None = None, message_type: Literal['TextMESSAGE', 'VoiceMESSAGE', 'DirectMESSAGE'] | None = None, sort_by: Literal['successful', 'failed', 'guild_snow', 'guild_name', 'author_snow', 'author_name'] = 'successful', sort_by_direction: Literal['asc', 'desc'] = 'desc', limit: int = 500, group_by: Literal['year', 'month', 'day'] = 'day'*) → list

Counts all the messages in the configured group based on parameters.

Parameters

- **guild** (*int*) – The snowflake id of the guild.
- **author** (*int*) – The snowflake id of the author.
- **after** (*Union[datetime, None] = None*) – Only count messages sent after the datetime.
- **before** (*Union[datetime, None]*) – Only count messages sent before the datetime.
- **guild_type** (*Literal["USER", "GUILD"] | None,*) – Type of guild.
- **message_type** (*Literal["TextMESSAGE", "VoiceMESSAGE", "DirectMESSAGE"] | None,*) – Type of message.
- **sort_by** (*Literal["successful", "failed", "guild_snow", "guild_name", "author_snow", "author_name"],*) – Sort items by selected. Defaults to "successful"
- **sort_by_direction** (*Literal["asc", "desc"]*) – Sort items by `sort_by` in selected direction (asc = ascending, desc = descending). Defaults to "desc"
- **limit** (*int = 500*) – Limit of the rows to return. Defaults to 500.
- **group_by** (*Literal["year", "month", "day"]*) – Results returned are grouped by `group_by`

Returns

List of tuples.

Each tuple contains:

- Date
- Successful sends
- Failed sends
- Guild snowflake id,
- Guild name
- Author snowflake id,
- Author name

Return type
list[tuple[date, int, int, int, str, int, str]]

`async initialize()`

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 = '/home/docs/daf/History', delimiter: str = ';', fallback: LoggerBASE | None = None*`)`

New in version v2.2.

Caution: Invite link tracking is not supported with LoggerCSV!

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. Defaults to /<user-home>/daf/History
- **delimiter** (*str*) – The delimiter between columns to use. Defaults to ';'
- **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 delete_logs`(*table: Any*, *logs: List[Any]*)

Method used to delete log objects objects.

Parameters

- **table** (*Any*) – The logging table to delete from.
- **primary_keys** (*List[int]*) – List of Primary Key IDs that match the rows of the table to delete.

`async analytic_get_num_messages`(*guild: int | None = None*, *author: int | None = None*, *after: datetime = datetime.datetime(1, 1, 1, 0, 0)*, *before: datetime = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)*, *guild_type: Literal['USER', 'GUILD'] | None = None*, *message_type: Literal['TextMESSAGE', 'VoiceMESSAGE', 'DirectMESSAGE'] | None = None*, *sort_by: Literal['successful', 'failed', 'guild_snow', 'guild_name', 'author_snow', 'author_name'] = 'successful'*, *sort_by_direction: Literal['asc', 'desc'] = 'desc'*, *limit: int = 500*, *group_by: Literal['year', 'month', 'day'] = 'day'*) → list

Counts all the messages in the configured group based on parameters.

Parameters

- **guild** (*int*) – The snowflake id of the guild.
- **author** (*int*) – The snowflake id of the author.
- **after** (*Union[datetime, None] = None*) – Only count messages sent after the datetime.
- **before** (*Union[datetime, None]*) – Only count messages sent before the datetime.
- **guild_type** (*Literal["USER", "GUILD"] | None,*) – Type of guild.
- **message_type** (*Literal["TextMESSAGE", "VoiceMESSAGE", "DirectMESSAGE"] | None,*) – Type of message.
- **sort_by** (*Literal["successful", "failed", "guild_snow", "guild_name", "author_snow", "author_name"],*) – Sort items by selected. Defaults to "successful"
- **sort_by_direction** (*Literal["asc", "desc"]*) – Sort items by `sort_by` in selected direction (asc = ascending, desc = descending). Defaults to "desc"
- **limit** (*int = 500*) – Limit of the rows to return. Defaults to 500.
- **group_by** (*Literal["year", "month", "day"]*) – Results returned are grouped by `group_by`

Returns

List of tuples.

Each tuple contains:

- Date
- Successful sends
- Failed sends

- Guild snowflake id,
- Guild name
- Author snowflake id,
- Author name

Return type
list[tuple[date, int, int, int, str, int, str]]

`async initialize()`
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: str | None = None, password: str | None = None, server: str | None = None, port: int | None = None, database: str | None = None, dialect: Literal['sqlite', 'mssql', 'postgresql', 'mysql'] = None, fallback: LoggerBASE | None = Ellipsis)`

Changed in version v2.7:

- Invite link tracking.
- Default database file output set to /<user-home-dir>/daf/messages

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).
- `ModuleNotFoundError` – Extra requirements are required.

`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 analytic_get_num_messages`(*guild: int | None = None, author: int | None = None, after: datetime = datetime.datetime(1, 1, 1, 0, 0), before: datetime = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999), guild_type: Literal['USER', 'GUILD'] | None = None, message_type: Literal['TextMESSAGE', 'VoiceMESSAGE', 'DirectMESSAGE'] | None = None, sort_by: Literal['successful', 'failed', 'guild_snow', 'guild_name', 'author_snow', 'author_name'] = 'successful', sort_by_direction: Literal['asc', 'desc'] = 'desc', limit: int = 500, group_by: Literal['year', 'month', 'day'] = 'day'*) → List[Tuple[date, int, int, int, str, int, str]]

Counts all the messages in the configured group based on parameters.

Parameters

- **guild** (*int*) – The snowflake id of the guild.
- **author** (*int*) – The snowflake id of the author.
- **after** (*Union[datetime, None] = None*) – Only count messages sent after the datetime.
- **before** (*Union[datetime, None]*) – Only count messages sent before the datetime.
- **guild_type** (*Literal["USER", "GUILD"] | None,*) – Type of guild.
- **message_type** (*Literal["TextMESSAGE", "VoiceMESSAGE", "DirectMESSAGE"] | None,*) – Type of message.
- **sort_by** (*Literal["successful", "failed", "guild_snow", "guild_name", "author_snow", "author_name"],*) – Sort items by selected. Defaults to "successful"
- **sort_by_direction** (*Literal["asc", "desc"]*) – Sort items by `sort_by` in selected direction (asc = ascending, desc = descending). Defaults to "desc"
- **limit** (*int = 500*) – Limit of the rows to return. Defaults to 500.
- **group_by** (*Literal["year", "month", "day"]*) – Results returned are grouped by `group_by`

Returns

List of tuples.

Each tuple contains:

- Date
- Successfule sends

- Failed sends
- Guild snowflake id,
- Guild name
- Author snowflake id,
- Author name

Return type
list[tuple[date, int, int, int, str, int, str]]

Raises
`SQLAlchemyError` – There was a problem with the database.

analytic_get_message_log(*guild: int | None = None, author: int | None = None, after: datetime = datetime.datetime(1, 1, 1, 0, 0), before: datetime = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999), success_rate: Tuple[float, float] = (0, 100), guild_type: Literal['USER', 'GUILD'] | None = None, message_type: Literal['TextMESSAGE', 'VoiceMESSAGE', 'DirectMESSAGE'] | None = None, sort_by: Literal['timestamp', 'success_rate'] = 'timestamp', sort_by_direction: Literal['asc', 'desc'] = 'desc', limit: int = 500*) → List[MessageLOG]

Returns a list of *MessageLOG* objects (message logs) that match the parameters.

Parameters

- **guild** (*Union[int, None]*) – The snowflake id of the guild.
- **author** (*Union[int, None]*) – The snowflake id of the author.
- **after** (*Union[datetime, None]*) – Include only messages sent after this datetime.
- **before** (*Union[datetime, None]*) – Include only messages sent before this datetime.
- **success_rate** (*tuple[int, int]*) – Success rate tuple containing minimum success rate and maximum success rate the message log can have for it to be included. Success rate is measured in % and it is defined by:

  Successfully sent channels / all channels.
- **guild_type** (*Literal["USER", "GUILD"] | None,*) – Type of guild.
- **message_type** (*Literal["TextMESSAGE", "VoiceMESSAGE", "DirectMESSAGE"] | None,*) – Type of message.
- **sort_by** (*Literal["timestamp", "success_rate", "data"],*) – Sort items by selected. Defaults to "timestamp"
- **sort_by_direction** (*Literal["asc", "desc"] = "desc"*) – Sort items by `sort_by` in selected direction (asc = ascending, desc = descending). Defaults to "desc"
- **limit** (*int = 500*) – Limit of the message logs to return. Defaults to 500.

Returns
List of the message logs.

Return type
list[MessageLOG]

async analytic_get_num_invites(*guild: int | None = None, after: datetime = datetime.datetime(1, 1, 1, 0, 0), before: datetime = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999), sort_by: Literal['count', 'guild_snow', 'guild_name', 'invite_id'] = 'count', sort_by_direction: Literal['asc', 'desc'] = 'desc', limit: int = 500, group_by: Literal['year', 'month', 'day'] = 'day'*) → List[Tuple[date, int, str]]

Returns invite link join counts.

Parameters

- **guild** (*int*) – The snowflake id of the guild.
- **after** (*Union[datetime, None] = None*) – Only count messages sent after the datetime.
- **before** (*Union[datetime, None]*) – Only count messages sent before the datetime.
- **sort_by** (*Literal["count", "guild_snow", "guild_name", "invite_id"],*) – Sort items by selected. Defaults to "successful"
- **sort_by_direction** (*Literal["asc", "desc"]*) – Sort items by `sort_by` in selected direction (asc = ascending, desc = descending). Defaults to "desc"
- **limit** (*int = 500*) – Limit of the rows to return. Defaults to 500.
- **group_by** (*Literal["year", "month", "day"]*) – Results returned are grouped by `group_by`

Returns

List of tuples.

Each tuple contains:

- Date
- Invite count
- Invite ID

Return type

list[Tuple[date, int, int, str, str]]

Raises

SQLAlchemyError – There was a problem with the database.

analytic_get_invite_log(*guild: int | None = None, invite: str | None = None, after: datetime = datetime.datetime(1, 1, 1, 0, 0), before: datetime = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999), sort_by: Literal['timestamp'] = 'timestamp', sort_by_direction: Literal['asc', 'desc'] = 'desc', limit: int = 500*) → List[InviteLOG]

Returns a list of *InviteLOG* objects (invite logs) filtered by the given parameters.

Parameters

- **guild** (*Union[int, None]*) – The snowflake id of the guild.
- **invite** (*Union[str, None]*) – Discord invite ID (final part of URL).
- **after** (*Union[datetime, None]*) – Include only invites sent after this datetime.
- **before** (*Union[datetime, None]*) – Include only invites sent before this datetime.
- **sort_by** (*Literal["timestamp"],*) – Sort items by selected. Defaults to "timestamp"

- **sort_by_direction** (*Literal["asc", "desc"] = "desc"*) – Sort items by sort_by in selected direction (asc = ascending, desc = descending). Defaults to "desc"
- **limit** (*int = 500*) – Limit of the invite logs to return. Defaults to 500.

Returns
List of the message logs.

Return type
list[InviteLOG]

`async delete_logs`(*logs: List[MessageLOG | InviteLOG]*)

Method used to delete log objects objects.

Parameters

- **table** (*MessageLOG | InviteLOG*) – The logging table to delete from.
- **primary_keys** (*List[int]*) – List of Primary Key IDs that match the rows of the table to delete.

`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.

Message data

FILE

`class daf.messagedata.file.FILE`(*filename: str, data: bytes | str | None = None*)

FILE object used as a data parameter to the xMESSAGE 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.

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

Parameters

- **filename** (*str*) – The filename of file you want to send.
- **data** (*Optional[Union[bytes, str]]*) – Optional raw data or hex string representing raw data.

If this parameter is not given (set as None), the data will be automatically obtained from `filename` file. Defaults to `None`.

New in version 2.10.

Raises

- `FileNotFoundError` – The `filename` does not exist.
- `OSError` – Could not read file `filename`.
- `ValueError` – The `data` parameter is of incorrect format.

`property stream: BytesIO`

Returns a stream to data provided at creation.

`property filename: str`

The name of the file

`property fullpath: str`

The full path to the file

`property data: bytes`

Returns the raw binary data

`property hex: str`

Returns HEX representation of the data.

`to_dict()`

Returns dictionary representation of this data type.

New in version 2.10.

TextMessageData

class `daf.messagedata.TextMessageData`(*content: str | None = None*, *embed: ~_discord.embeds.Embed | None = None*, *files: ~typing.List[~daf.messagedata.file.FILE] = <factory>*)

Represents fixed text message data.

VoiceMessageData

class `daf.messagedata.VoiceMessageData`(*file:* FILE)

Represents fixed voice-like data.

DynamicMessageData

class `daf.messagedata.DynamicMessageData`

Represents dynamic message data. Can be both text or voice, but make sure text is only used on *`TextMESSAGE`* and voice on *`VoiceMESSAGE`*.

This needs to be inherited and the subclass needs to implement the `get_data` method, which accepts no parameters (pass those through the class).

Example

```python
class MyCustomText(DynamicMessageData):
    def __init__(self, a: int):
        self.a = a

    def get_data(self):  # Can also be async
        return TextMessageData(f"Passed parameter was: {self.a}")

class MyCustomVoice(DynamicMessageData):
    def get_data(self):  # Can also be async
        return VoiceMessageData(FILE("./audio.mp3"))


TextMESSAGE(data=MyCustomText(152))
VoiceMESSAGE(data=MyCustomVoice())
```

abstract get_data() → BaseMessageData

The data getter method. Needs to be implemented in a subclass.

The method must return either a *TextMessageData* or a *VoiceMessageData* instance. It can also return None if no data is to be sent.

Text matching (logic)

and

class daf.logic.**and_**(**args*, *operands: List[BaseLogic] = []*)

Represents logical *AND* operator.

Parameters

args (*Unpack[BaseLogic]*) – Arbitrary number of operands (either logic boolean or text operands).

or

class daf.logic.**or_**(**args*, *operands: List[BaseLogic] = []*)

Represents logical *OR* operator.

Parameters

args (*Unpack[BaseLogic]*) – Arbitrary number of operands (either logic boolean or text operands).

not

class daf.logic.**not_**(*operand: BaseLogic*)

Represents logical *NOT* operator.

Parameters
operand (*BaseLogic*) – A single operand to negate.

contains

class daf.logic.**contains**(*keyword: str*)

Text matching condition.

Parameters
keyword (*str*) – The keyword needed to be inside a text message.

regex

class daf.logic.**regex**(*pattern: str, flags: RegexFlag = RegexFlag.MULTILINE, full_match: bool = False*)

RegEx (regular expression) text matching condition.

Parameters

- **pattern** (*str*) – The RegEx pattern string.
- **flags** (*Optional[re.RegexFlag]*) – RegEx (binary) flags. Defaults to re.MULTILINE.
- **full_match** (*Optional[bool]*) – Boolean parameter. If True, the `pattern` must capture the entire text message string for a match. Defaults to False.

Auto objects

AutoCHANNEL

class daf.message.**AutoCHANNEL**(*include_pattern: BaseLogic | str, exclude_pattern: str | None = None*)

New in version v2.3.

Changed in version v2.10: *daf.message.AutoCHANNEL.remove()* will now prevent the channel from being added again.

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 5.7: Usage

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

Parameters
include_pattern (*BaseLogic*) – Matching condition, which checks if the name of channels matches defined condition.

property channels: List[TextChannel | Thread | VoiceChannel]

Return a list of found channels

async initialize(*parent*, *channel_getter: Callable*)

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

Changed in version v2.10: Changed the channel `channel_type` into `channel_getter`, which is now a function that can be used to get a list of all the correct channels.

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: TextChannel | Thread | 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(*init_options=None*, ***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: BaseLogic | str*, *exclude_pattern: str | None = None*, *remove_after: timedelta | datetime | None = None*, *messages: List[BaseChannelMessage] | None = None*, *logging: bool | None = False*, *auto_join: GuildDISCOVERY | None = None*, *invite_track: List[str] | None = None*, *removal_buffer_length: int = 50*)

Changed in version v4.0.0:

- Restored original way AutoGUILD works (as a GUILD generator)
- Include and exclude patterns now accept a daf.logic.BaseLogic object for more flexible matching. Using strings (text) is deprecated and will be removed.

Represents multiple guilds (servers) based on a text pattern. AutoGUILD generates `daf.guild.GUILD` objects for each matched pattern.

Parameters

- **include_pattern** (*BaseLogic*) – Matching condition, which checks if the name of guilds matches defined condition.
- **remove_after** (*Optional[Union[timedelta, datetime]] = None*) – When to remove this object from the shilling list.

- **messages** (*List[BaseChannelMessage]*) – List of messages with channels. This includes TextMESSAGE and VoiceMESSAGE.
- **logging** (*Optional[bool] = False*) – Set to True if you want the guilds generated to log sent messages.
- **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.
- **invite_track** (*Optional[List[str]]*) – List of invite links (or invite codes) to track.
- **removal_buffer_length** (*int*) – The size of removed messages buffer. A message is added to this buffer when remove_message is called by the user or a message automatically removes itself for any reason. Defaults to 50.

property messages: List[BaseChannelMessage]

New in version 3.0.

Returns all the (template) message objects.

property guilds: List[*GUILD*]

Returns cached GUILD objects.

property remove_after: datetime | None

Returns the timestamp at which AutoGUILD will be removed or None if it will never be removed.

add_message(*message: BaseChannelMessage*) → Future

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).

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Parameters
message (*BaseChannelMessage*) – Message object to add.

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

Raises

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

remove_message(*message: BaseChannelMessage*) → Future

Remove a `message` from the advertising list.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

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

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type

Awaitable

`update`(*init_options=None*, ***kwargs*) → Future

Updates the object with new initialization parameters.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Returns

- *Awaitable* – An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.
- .. *WARNING::* – After calling this method the entire object is reset (this includes it's GUILD objects in cache).

`async initialize`(*parent: Any*, *event_ctrl: EventController*)

Initializes the object.

Message period

FixedDurationPeriod

`class daf.message.messageperiod.FixedDurationPeriod`(*duration: timedelta*, *next_send_time: timedelta | datetime | None = None*)

A fixed message (sending) period.

Parameters

- **duration** (*timedelta*) – The period duration (how much time to wait after every send).
- **next_send_time** (*datetime | timedelta*) – Represents the time at which the message should first be sent. Use `datetime` to specify the exact date and time at which the message should start being sent. Use `timedelta` to specify how soon (after creation of the object) the message should start being sent.

`adjust`(*minimum: timedelta*) → None

Adjust the period to always be greater than the `minimum`.

`calculate`()

Calculates the next datetime the message is going to be sent.

`defer`(*dt: datetime*)

Defers advertising until `dt` This should be used for overriding the normal next datetime the message was supposed to be sent on.

`get`() → datetime

Returns the next datetime the message is going to be sent.

RandomizedDurationPeriod

class `daf.message.messageperiod.`**`RandomizedDurationPeriod`**(*minimum: timedelta*, *maximum: timedelta*, *next_send_time: timedelta | datetime | None = None*)

A randomized message (sending) period. After every send, the message will wait a different randomly chosen period within `minimum` and `maximum`.

Parameters

- **`minimum`** (*`timedelta`*) – Bottom limit of the randomized period.
- **`maximum`** (*`timedelta`*) – Upper limit of the randomized period.
- **`next_send_time`** (*`datetime | timedelta`*) – Represents the time at which the message should first be sent. Use `datetime` to specify the exact date and time at which the message should start being sent. Use `timedelta` to specify how soon (after creation of the object) the message should start being sent.

`adjust`(*minimum: timedelta*) → None

Adjust the period to always be greater than the `minimum`.

`calculate`()

Calculates the next datetime the message is going to be sent.

`defer`(*dt: datetime*)

Defers advertising until `dt` This should be used for overriding the normal next datetime the message was supposed to be sent on.

`get`() → datetime

Returns the next datetime the message is going to be sent.

DaysOfWeekPeriod

class `daf.message.messageperiod.`**`DaysOfWeekPeriod`**(*days: list[Literal['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']]*, *time: time*, *next_send_time: timedelta | datetime | None = None*)

Represents a period that will send on `days` at specific `time`.

E. g., parameters `days=["Mon", "Wed"]` and `time=time(hour=12, minute=0)` produce a behavior that will send a message every Monday and Wednesday at 12:00.

Parameters

- **`days`** (*`list[Literal["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]]`*) – List of day abbreviations on which the message will be sent.
- **`time`** (`datetime.time`) – The time on which the message will be sent (every day of `days`).
- **`next_send_time`** (*`datetime | timedelta`*) – Represents the time at which the message should first be sent. Use `datetime` to specify the exact date and time at which the message should start being sent. Use `timedelta` to specify how soon (after creation of the object) the message should start being sent.

Raises

`ValueError` – The `days` parameter was an empty list.

`calculate()`

Calculates the next datetime the message is going to be sent.

`adjust`(*minimum: timedelta*) → None

Adjust the period to always be greater than the `minimum`.

`defer`(*dt: datetime*)

Defers advertising until `dt` This should be used for overriding the normal next datetime the message was supposed to be sent on.

`get`() → datetime

Returns the next datetime the message is going to be sent.

DailyPeriod

`class daf.message.messageperiod.DailyPeriod`(*time: time, next_send_time: timedelta | datetime | None = None*)

Represents a daily send period. Messages will be sent every day at `time`.

Parameters

- **`time`** (*`time`*) – The time on which the message will be sent.
- **`next_send_time`** (*`datetime | timedelta`*) – Represents the time at which the message should first be sent. Use `datetime` to specify the exact date and time at which the message should start being sent. Use `timedelta` to specify how soon (after creation of the object) the message should start being sent.

`adjust`(*minimum: timedelta*) → None

Adjust the period to always be greater than the `minimum`.

`calculate()`

Calculates the next datetime the message is going to be sent.

`defer`(*dt: datetime*)

Defers advertising until `dt` This should be used for overriding the normal next datetime the message was supposed to be sent on.

`get`() → datetime

Returns the next datetime the message is going to be sent.

Messages

TextMESSAGE

`class daf.message.TextMESSAGE`(*start_period: timedelta | int | None = None, end_period: int | timedelta = None, data: BaseTextData | list | tuple | set | str | Embed | FILE | _FunctionBaseCLASS = None, channels: list[Union[int, _discord.channel.TextChannel, _discord.threads.Thread]] | AutoCHANNEL = None, mode: Literal['send', 'edit', 'clear-send'] = 'send', start_in: timedelta | datetime | None = None, remove_after: int | timedelta | datetime | None = None, auto_publish: bool = False, period: BaseMessagePeriod = None*)

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

Parameters

- **data** (*BaseTextData*) – The data to be sent. Can be TextMessageData or class inherited from DynamicMessageData
- **channels** (*Union[list[Union[int, discord.TextChannel, discord.Thread]], daf.message.AutoCHANNEL]*) – Channels that it will be advertised into (Can be snowflake ID or channel objects from PyCord).

  Changed in version v2.3: Can also be *AutoCHANNEL*

  Note: If no channels are left, the message is automatically removed, unless AutoCHANNEL is used.

- **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.
- **remove_after** (*Optional[Union[int, timedelta, datetime]]*) – Deletes the message after:
  - int - provided amounts of successful sends to seperate channels.
  - timedelta - the specified time difference
  - datetime - specific date & time

  Changed in version 2.10: Parameter `remove_after` of int type will now work at a channel level and it nows means the SUCCESSFUL number of sends into each channel.
- **auto_publish** (*Optional[bool]*) – Automatically publish message if sending to an announcement channel. Defaults to False.

  If the channel publish is rate limited, the message will still be sent, but an error will be printed to the console instead of message being published to the follower channels.

  New in version 2.10.

generate_log_context(*content: str | None, embed: Embed, files: List[FILE], succeeded_ch: List[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*, *event_ctrl: EventController*, *channel_getter: Callable*)

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

property remove_after: int | datetime | None

Returns the remaining send counts / date after which the message will be removed from the sending list. If the original type of the `remove_after` parameter to the message was of type int, this will return the maximum remaining amount of sends from all channels. If all the channels have been removed, this will return the original count `remove_after` parameter.

update(*_init_options: dict | None = None*, ***kwargs: Any*) → Future

New in version v2.0.

Changed in version v3.0: Turned into async api.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

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.

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

Raises

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

DirectMESSAGE

class daf.message.**DirectMESSAGE**(*start_period: int | timedelta | None = None, end_period: int | timedelta = None, data: BaseTextData | list | tuple | set | str | Embed | FILE | _FunctionBaseCLASS = None, mode: Literal['send', 'edit', 'clear-send'] | None = 'send', start_in: timedelta | datetime | None = None, remove_after: int | timedelta | datetime | None = None, period: BaseMessagePeriod = None*)

This class is used for creating objects that represent messages which will be sent to user's private messages.

Deprecated since version v2.1:

- start_period, end_period - Using int values, use `timedelta` object instead.

Changed in version v2.7: *start_in* now accepts datetime object

Parameters

- **data** (*BaseTextData*) – The data to be sent. Can be TextMessageData or class inherited from DynamicMessageData
- **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.
- **remove_after** (*Optional[Union[int, timedelta, datetime]]*) – Deletes the guild after:
  - int - provided amounts of successful sends
  - timedelta - the specified time difference
  - datetime - specific date & time

generate_log_context(*success_context: Dict[str, bool | Exception | None], content: str | None, embed: Embed | None, 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*, *event_ctrl: EventController*, *guild: User*)

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

property remove_after: Any

New in version v3.0.

Returns the remaining send counts / date after which the message will be removed from the sending list.

update(*_init_options: dict* | *None* = *None*, ***kwargs*) → Future

New in version v2.0.

Changed in version v3.0: Turned into async api.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

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.

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type

Awaitable

Raises

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

VoiceMESSAGE

class daf.message.VoiceMESSAGE(*start_period: int | timedelta | None = None*, *end_period: int | timedelta = None*, *data: BaseVoiceData | FILE | list | tuple | set | _FunctionBaseCLASS = None*, *channels: list[Union[int, _discord.channel.VoiceChannel]] | AutoCHANNEL = None*, *volume: int | None = 50*, *start_in: timedelta | datetime | None = None*, *remove_after: int | timedelta | datetime | None = None*, *period: BaseMessagePeriod = None*)

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

Warning: This additionally requires FFMPEG to be installed on your system.

Parameters

- **data** (*BaseVoiceData*) – The actual data streamed to voice channels. Can be VoiceMessageData or a class inherited from DynamicMessageData.
- **channels** (*Union[list[Union[int, discord.VoiceChannel]], daf.message.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
- **remove_after** (*Optional[Union[int, timedelta, datetime]]*) – Deletes the message after:
  - int - provided amounts of successful sends to separate channels.
  - timedelta - the specified time difference
  - datetime - specific date & time
- **period** (*BaseMessagePeriod*) – The sending period.

generate_log_context(*file: FILE*, *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:
    {
        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]

property remove_after: int | datetime | None

Returns the remaining send counts / date after which the message will be removed from the sending list. If the original type of the `remove_after` parameter to the message was of type int, this will return the maximum remaining amount of sends from all channels. If all the channels have been removed, this will return the original count `remove_after` parameter.

update(*_init_options: dict* | *None* = *None*, ***kwargs: Any*) → Future

New in version v2.0.

Changed in version v3.0: Turned into async api.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

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.

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

Raises

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

`async initialize`(*parent: Any*, *event_ctrl: EventController*, *channel_getter: Callable*)

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

Guilds

GUILD

`class daf.guild.GUILD`(*snowflake: int | Guild*, *messages: List[BaseChannelMessage] | None = None*, *logging: bool | None = False*, *remove_after: timedelta | datetime | None = None*, *invite_track: List[str] | None = None*, *removal_buffer_length: int = 50*)

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

Changed in version v3.0:

- Removed `created_at` property.
- New `remove_after` property

Changed in version v2.7: Added `invite_track` 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
- **`invite_track`** (`Optional[List[str]]`) – New in version 2.7.

  List of invite IDs to be tracked for member join count inside the guild. **Bot account** only, does not work on user accounts.

Note: Accounts are required to have *Manage Channels* and *Manage Server* permissions inside a guild for tracking to fully function. *Manage Server* is needed for getting information about invite links, *Manage Channels* is needed to delete the invite from the list if it has been deleted, however tracking still works without it.

Warning: For GUILD to receive events about member joins, `members` intent is required to be True inside the `intents` parameters of *`daf.client.ACCOUNT`*. This is a **privileged intent** that also needs to be enabled though Discord's developer portal for each bot. After it is enabled, you can set it to True .

Invites intent is also needed. Enable it by setting `invites` to True inside the `intents` parameter of *`ACCOUNT`*.

- **`removal_buffer_length`** (*`Optional[int]`*) – Maximum number of messages to keep in the removed_messages buffer.

  New in version 3.0.

`async initialize`(*parent: Any*, *event_ctrl: EventController*) → 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.

`add_message`(*message: TextMESSAGE* | *VoiceMESSAGE*)

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).

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

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

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

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

`generate_invite_log_context`(*member: Member*, *invite_id: str*) → dict

Generates dictionary representing the log of a member joining a guild.

Parameters
`member` (*`discord.Member`*) – The member that joined a guild.

Returns

```
{
    "id": ID of the invite,
    "member": {
        "id": Member ID,
        "name": Member name
    }
}
```

Return type
dict

property apiobject: Object

New in version v2.4.

Returns the Discord API wrapper's object of self.

generate_log_context() → Dict[str, 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.

property remove_after: datetime | None

Returns the timestamp at which object will be removed or None if it will not be removed.

remove_message(*message: BaseMESSAGE*) → Future

Removes a message from the message list.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Changed in version 3.0: The function is now async.

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

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

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 removed_messages: List[BaseMESSAGE]

Returns a list of messages that were removed from server (last `removal_buffer_length` messages).

property snowflake: int

New in version v2.0.

Returns the discord's snowflake ID.

update(*init_options=None*, ***kwargs*) → Future

New in version v2.0.

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

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

> **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.

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type

Awaitable

Raises

`Union[TypeError, ValueError]` – Invalid keyword argument was passed.

USER

class daf.guild.USER(*snowflake: int | User*, *messages: List[DirectMESSAGE] | None = None*, *logging: bool | None = False*, *remove_after: timedelta | datetime | None = None*, *removal_buffer_length: int = 50*)

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

Changed in version v3.0:

- Removed `created_at` property.
- New `remove_after` property

Changed in version v2.7: Added `invite_track` 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

- `removal_buffer_length` (*Optional[int]*) – Maximum number of messages to keep in the removed_messages buffer.

  New in version 3.0.

`add_message`(*message: DirectMESSAGE*) → Future

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).

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

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

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

Raises

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

`async initialize`(*parent: Any*, *event_ctrl: EventController*)

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

`property apiobject: Object`

New in version v2.4.

Returns the Discord API wrapper's object of self.

`generate_log_context`() → Dict[str, 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.

`property remove_after: datetime | None`

Returns the timestamp at which object will be removed or None if it will not be removed.

`remove_message`(*message: BaseMESSAGE*) → Future

Removes a message from the message list.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Changed in version 3.0: The function is now async.

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

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

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 removed_messages: List[BaseMESSAGE]

Returns a list of messages that were removed from server (last `removal_buffer_length` messages).

property snowflake: int

New in version v2.0.

Returns the discord's snowflake ID.

update(*init_options=None*, ***kwargs*) → Future

New in version v2.0.

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

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

> **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.

Returns
An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type
Awaitable

Raises
`Union[TypeError, ValueError]` – Invalid keyword argument was passed.

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.

async `initialize()` → None

Starts the webdriver whenever the framework is started.

Returns
`True` on success or `False` on error.

Return type
bool

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 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: str | None = None, is_user: bool | None = False, intents: Intents | None = None, proxy: str | None = None, servers: List[GUILD | USER | AutoGUILD] | None = None, username: str | None = None, password: str | None = None, removal_buffer_length: int = 50, responders: List[ResponderBase] = None*)

New in version v2.4.

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

Changed in version v3.0: When servers are added directly through account initialization, they will be removed upon errors and available withing `removed_servers` property.

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 (settings of events that the client will subscribe to). Defaults to everything enabled except `members`, `presence` and `message_content`, as those are privileged events, which need to be enabled though Discord's developer settings for each bot.

> **Warning:** For invite link tracking to work, it is required to set `members` intents to True. Invites intent is also needed. Enable it by setting `invites` to True inside the `intents` parameter of *ACCOUNT*.
>
> Intent `guilds` is also required for AutoGUILD and AutoCHANNEL, however it is automatically forced to True, as it is not a privileged intent.

- **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). If initializing a server fails (eg. server doesn't exist on Discord), it will be removed and added to *daf.client.ACCOUNT.removed_servers* property.
- **username** (*Optional[str]*) – The username to login with.
- **password** (*Optional[str]*) – The password to login with.
- **removal_buffer_length** (*Optional[int]*) – New in version 3.0.

  Maximum number of servers to keep in the removed_servers buffer.
- **responders** (*List[responder.ResponderBase]*) – New in version 3.3.

  List of automatic responders. These will automatically respond to certain messages.

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: List[*GUILD* | *USER* | *AutoGUILD*]

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

property removed_servers: List[*GUILD* | *USER* | *AutoGUILD*]

Returns a list of servers that were removed from account (last `removal_buffer_length` servers).

property client: Client

Returns the API wrapper client

property responders

Returns the list of automatic message responders

add_server(*server:* GUILD | USER | AutoGUILD) → Future

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

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Parameters

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

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type

Awaitable

Raises

- **ValueError** – Invalid `snowflake` (eg. server doesn't exist).
- **RuntimeError** – Could not query Discord.

remove_server(*server:* GUILD | USER | AutoGUILD) → Future

Removes a guild like object from the shilling list.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Changed in version 3.0: Removal is now asynchronous.

Parameters

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

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type

Awaitable

Raises

ValueError – `server` is not in the shilling list.

get_server(*snowflake: int* | *Guild* | *User* | *Object*) → *GUILD* | *USER* | None

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.

`add_responder`(*resp: ResponderBase*) → Future

New in version 3.3.0.

Adds an automatic message responder to the account.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Parameters

resp (*ResponderBase*) – Any inherited class of the `daf.responder.ResponderBase` interface.

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await account.add_responder()`.

Return type

Awaitable

`remove_responder`(*resp: ResponderBase*) → Future

New in version 3.3.0.

Removes an automatic message responder from the account.

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

Parameters

resp (*ResponderBase*) – Any inherited class of the `daf.responder.ResponderBase` interface.

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await account.remove_responder()`.

Return type

Awaitable

`update`(***kwargs*) → Future

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

This is an asynchronous API operation. When returning from this function, the action is not immediately executed.

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

Returns

An awaitable object which can be used to await for execution to finish. To wait for the execution to finish, use `await` like so: `await method_name()`.

Return type

Awaitable

Raises

- `ValueError` – The account is no longer running.
- `Exception` – Other exceptions returned from *`daf.client.ACCOUNT`* constructor or from *`daf.client.ACCOUNT.initialize()`*.

`async initialize()`

Initializes the API wrapper client layer.

`generate_log_context()` → Dict[str, str | int]

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

Return type

Dict[str, Union[str, int]]

RemoteAccessCLIENT

`class daf.remote.RemoteAccessCLIENT`(*host: str | None = '0.0.0.0', port: int | None = 80, username: str | None = None, password: str | None = None, certificate: str | None = None, private_key: str | None = None, private_key_pwd: str | None = None*)

Client used for processing remote requests from a GUI located on a different network.

Parameters

- **`host`** (*`Optional[str]`*) – The host address. Defaults to `0.0.0.0` (Listens on all network interfaces).
- **`port`** (*`Optional[int]`*) – The http port. Defaults to `80`.
- **`username`** (*`Optional[str]`*) – The basic authorization username. Defaults to `None`.
- **`password`** (*`Optional[str]`*) – The basic authorization password. Defaults to `None`.
- **`certificate`** (*`Optional[str]`*) – Path to a certificate file. Used when HTTPS is desired instead of HTTP. (Recommended if username & password)
- **`private_key`** (*`Optional[str]`*) – Path to a private key file that belongs to `certificate`.
- **`private_key_pwd`** (*`Optional[str]`*) – The password of `private_key` if it has any.

Raises

`ValueError` – Private key is required with certificate.

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: 1>`

`B100_1k = <QueryMembers.B100_1k: 2>`

`B1k_10k = <QueryMembers.B1k_10k: 3>`

`ABV_10k = <QueryMembers.ABV_10k: 4>`

GuildDISCOVERY

class `daf.web.`**`GuildDISCOVERY`**(*prompt: str*, *sort_by: QuerySortBy | None = QuerySortBy.TOP*, *total_members: QueryMembers | None = QueryMembers.ALL*, *limit: int | None = 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.

async **`initialize`**(*parent: Any*)

Initializes guild discovery session.

DAF control reference

initialize

async `daf.core.`**`initialize`**(*user_callback: Callable | Coroutine | None = None*, *debug: TraceLEVELS | int | str | None = TraceLEVELS.NORMAL*, *logger: LoggerBASE | None = None*, *accounts: List[ACCOUNT] | None = None*, *save_to_file: bool = False*, *remote_client: RemoteAccessCLIENT | None = None*) → 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() → None`

Stops and cleans the framework.

run

`daf.core.run(user_callback: Callable | Coroutine | None = None, debug: TraceLEVELS | int | str | bool | None = TraceLEVELS.NORMAL, logger: LoggerBASE | None = None, accounts: List[ACCOUNT] | None = None, save_to_file: bool = False, remote_client: RemoteAccessCLIENT | None = None) → None`

Changed in version 2.7: Removed deprecated parameters (see *v2.7*)

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

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.

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 with the `path` parameter set to /<user-home-dir>/daf/History
- **accounts** (*Optional[List[client.ACCOUNT]]*) – List of *ACCOUNT* (Discord accounts) to use. .. versionadded:: v2.4
- **save_to_file** (*Optional[bool]*) – If `True`, the shilling list (of accounts, guilds, messages, …) will be saved to file and preserved on shutdown.

  It is recommended you set this to `False` when passing `run()` or `initialize()` the statically defined `accounts` parameter.

  Note: Setting this to True and passing the `accounts` parameter as well, results in *Account already added* warnings.

Raises

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

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.guilduser.USER, daf.guild.guilduser.GUILD, daf.guild.autoguild.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.
- `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.guilduser.GUILD, daf.guild.guilduser.USER*]*) → None

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: 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.

5.2.2 HTTP reference

Contain classes and functions description of the HTTP API.

Connection

http_ping

`async daf.remote.http_ping()`

Pinging route for testing connection.

Route
/ping

Method
GET

Logging

http_get_logger

async daf.remote.**http_get_logger**()

Returns active message / invite logger.

Returns
Active logger.

Return type
LoggerBASE

Route
/logging

Method
GET

Object

http_get_object

async daf.remote.**http_get_object**(*object_id: int*)

Returns a tracked object, tracked with @track_id decorator.

Parameters
object_id (*int*) – The ID of the object to obtain.

Returns
The object linked to `object_id`.

Return type
object

Route
/object

Method
GET

http_execute_method

async daf.remote.**http_execute_method**(*object_id: int*, *method_name: str*, ***kwargs*)

Executes a method on a object. The method is an actual Python method.

Parameters

- **object_id** (*int*) – The ID of the object to execute on.
- **method_name** (*str*) – The name of the method to execute.
- **kwargs** – Variadic keyword arguments to pass to the executed method.

Returns
The returned value from method.

Return type
Any

Route
/method

Method
POST

http_get_accounts

async daf.core.**http_get_accounts**()

Retrieves all active accounts in the framework.

Returns
The active accounts.

Return type
List[*ACCOUNT*]

Route
/accounts

Method
GET

http_add_account

async daf.core.**http_add_account**(*account: dict*)

Adds a new account to the framework.

Parameters
account (ACCOUNT) – The account to initialize and add.

Route
/accounts

Method
POST

http_remove_account

async daf.core.**http_remove_account**(*account_id: int*)

Removes an account from the framework.

Parameters
account_id (*int*) – The ID of the account.

Route
/accounts

Method
DELETE

5.3 Changelog

5.3.1 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.

[Not yet released]
Documented changes are not yet available to use.

5.3.2 Releases

v4.0.5

- Fixed exe build.

v4.0.4

- Fixed automatic responder's not being removable over a remote connection.
- Fixed casting error when trying to update objects with `Literal` parameters.
- Other GUI fixes (from tkclasswiz library)

v4.0.3

- Fixed object editing window saving to an incorrect index (and removing other objects).

v4.0.2

- Fixed automatic responders not being serializable.

v4.0.1

- Fixed remote serialization of regex and time.

v4.0.0

- New automatic message responder (DM and guild) - *Automatic responder*.
- Changed message's parameters:
  - Deprecated:
    - `start_in`
    - `start_period`
    - `end_period`
    - All existing `data` types. They have been replaced with *`daf.messagedata.TextMessageData`* and *`daf.messagedata.DynamicMessageData`*
- Changed *`AutoGUILD`*'s parameters:
  - Deprecated:
    - `exclude_pattern`
    - Using `str` on `include_pattern`
  - Changed:
    - Made `include_pattern` accept *logic classes* (`and_`, `or_`, `contains`, ...)
- Changed *`AutoCHANNEL`*'s parameters:
  - Deprecated:
    - `exclude_pattern`
    - Using `str` on `include_pattern`
  - Changed:
    - Made `include_pattern` accept *logic classes* (`and_`, `or_`, `contains`, ...)
- Upon slow-mode / timeout, messages will now longer wait until the end of slow-mode / timeout. Instead, they will defer until the next period that is not within slow-mode / timeout. The period will however still auto-correct itself to be above the slow-mode.
- **[Breaking change]** Changed how *`daf.guild.AutoGUILD`* works. It will now create *`daf.guild.GUILD`* instances. *`daf.guild.AutoGUILD.guilds`* will also now return a list of generated *`daf.guild.GUILD`* objects instead of a list of `discord.Guild` objects. This also prevents a "bug" that appeared if the user was timed-out in a guild, which reflected upon other guilds as well. The added benefit of creating *`GUILD`* is different randomized sending periods across multiple guilds (assuming randomized sending period was configured).
- **[Breaking change]** Removed the deprecated `daf.dtypes.AUDIO`, which has been replaced with `daf.dtypes.FILE`.
- **[Potentially breaking change]** Changed event names
- GUI:
  - Theme selection support (top-left corner)

v3.2.2

- Fixed entire software not launching if SQL is not installed. The problem was some SQL classes were not defined.

v3.2.1

- GUI: Fixed conversion from GUI data to a Python daf core script.

v3.2.0

- GUI:
  - Moved library tkclasswiz to a separate library on PyPI and made it a requirement.
  - Object nicknaming (part of tkclasswiz)
  - Type nicknaming (part of tkclasswiz)
  - Fixed bug where the object edit window could not be closed after trying to edit a non-editable object.
- **[Breaking change]** Minimum Python version bumped to **Python 3.9**.

v3.1.2

- Fixed SQL compatibility
- Fixed "TypeError: can't compare offset-naive and offset-aware datetimes" exception when a rate limit happened (or slow mode).
- Fixed selenium timer reset when no join attempt was triggered.

v3.1.1

- Fixed guild and text channels not fully visible in property view of GUI.

v3.1.0

- Compatible with Python 3.12
- GUI:
  - ViewOnly structured data will display only the data that is provided, meaning the GUI will not be constructed based on type annotations of an objects, but rather based on the data itself.
  - Better toast notification format and compatibility across multiple DPI screens.
  - Graphical object library split into a separate package.
- *`daf.logging.LoggerJSON`*: - `index` field is now a unique snowflake-like ID (used for removing logs). - **[Breaking change]** Invite logs will now contain a "member" dictionary for each invite log. - Analytics are now supported.
- LoggerCSV: - Analytics are now supported. - `index` field added in order to allow removal of logs.
- **[Breaking change]** Removed long time deprecated package "framework", which was the original import.

v3.0.4

- Fixed AutoGUILD not working if the `messages` parameter is None.
- Fixed `verify_ssl` being ignored on the WebSocket connection.

v3.0.3

- Fixed "Loading from JSON template causes live object reference to be lost".

v3.0.2

- Fixed AutoGUILD not sending messages (events emitted prematurely).
- Fixed TextMESSAGE and VoiceMESSAGE not being removed after n sends when using AutoCHANNEL.
- Added missing `daf.guild.AutoGUILD.removed_messages` property.

v3.0.1

- Downgraded Selenium version from 4.13 to 4.12 since 4.13 does not support headless, which undetected-chromedriver is trying to set.

v3.0.0

- SQL analytics:
  - Counts now have better error reporting when an invalid value was passed.
- GUI:
  - Higher refresh rate due to threading redesign - instead of calling Tkinter's root.update inside an asyncio task, the root.mainroot is called directly, while the asyncio event loop is running inside another thread.
  - The GUI will not block the asyncio tasks (explained in previous bullet).
  - When saving a new object definition, if the type of a parameter is literal, the value will be pre-checked inside the GUI and an exception will be raised if a valid value is not given.
  - Properties that start with _ will no longer be displayed when viewing live structured objects.
  - Toast notifications for *`trace()`*.
  - Parameter validation for literals, enums and bool.
  - Copy / Paste globally for both drop-down menus and list menus.
- Core:
  - New events system and module
  - Updated PyCord API wrapper to 2.5.0 RC5
  - New property *`daf.client.ACCOUNT.removed_servers`* for tracking removed servers.
  - New property *`daf.guild.GUILD.removed_messages`* *`daf.guild.USER.removed_messages`* for tracking removed messages.
  - New parameter `removal_buffer_length` to *`daf.client.ACCOUNT`* for setting maximum amount of of servers to keep in the *`daf.client.ACCOUNT.removed_servers`* buffer.

  - New parameter removal_buffer_length to *daf.guild.GUILD* and *daf.guild.USER* for setting maximum amount of messages to keep in the *daf.guild.GUILD.removed_messages* / *daf.guild.USER.removed_messages* buffer.
  - Event loop based API - All API methods that get called now submit an event in the event loop, which causes the API call to happen asynchronously unless awaited with await keyword. This also makes DAF much more efficient.
  - Remote:
    * Persistent WebSocket connection for receiving events from the core server (eg. *trace()* events).
  - Removed remaining_before_removal property from all message classes.
  - Added remove_after property to *GUILD*, *USER*, *TextMESSAGE*, *VoiceMESSAGE* and *DirectMESSAGE*.

v2.10.4

- Fixed prematurely exiting when waiting for captcha to be completed by user.

v2.10.3

- Fixed Chrome driver not working with newer Chrome versions (115+).
- Fetching invite links better bypass.
- Remove invalid presence
- Fixed remaining_before_removal properties
- Fixed SQL queries not working on direct messages.

v2.10.2

- Fixed *Unclosed client session* warning when removing an user account.
- Fixed documentation of *daf.core.shutdown()* - removed information about non existent parameters.
- Selenium better waiting avoidance
- Fixed ACCOUNT not being removed from the list if the update failed and the re-login after update failed.

v2.10.1

- Fixed files in DirectMESSAGE.
- Fixed file paths over remote not having the full patch when returned back.
- Fixed files not having full path in the logs.
- Added daf.dtypes.FILE.fullpath to support the previous fix.
- Fixed exception when adding messages inside AutoGUILD, when one of the cached guilds fails initialization.
- Fixed serialization for discord.VoiceChannel, which included slowmode_delay, even though the attribute doesn't exist in the VoiceChannel.

v2.10

- GUI:
  - GUI can now be started with `python -m daf_gui`
  - Deprecation notices are now a button.
  - Certain fields are now masked with '*' when not editing the object.
  - Old data that is being updated will now be updated by index
  - View properties of trackable objects. This can be used to, eg. view the channels AutoCHANNEL found.
  - 'Load default' button when editing `discord.Intents` object.
  - A warning is shown besides the method execution frame to let users know, the data is not preserved.
  - Fixed accounts not being deleted when using delete / backspace keys in live view.
- Accounts:
  - Intents:
    - Added warnings for missing intents.
    - Intents.members is by default now disabled.
- Messages:
  - **[Breaking change]** Removed deprecated feature - YouTube streaming, in favor of faster startups and installation time.
  - New property: `remaining_before_removal`, `remaining_before_removal`, `remaining_before_removal`
  - New parameter: `auto_publish` to *`TextMESSAGE`* for automatically publishing messages sent to announcement (news) channels.
  - *`TextMESSAGE`* and *`VoiceMESSAGE`*'s `remove_after` parameter:
    - If integer, it will now work independently for each channel and will only decrement on successful sends.
    - If `datetime` or `timedelta`, it will work the same as before.
  - Moderation timeout handling (messages resume one minute after moderation timeout expiry)
  - Message content:
    - Deprecated `daf.dtypes.AUDIO`, replaced with `daf.dtypes.FILE`.
    - `daf.dtypes.FILE` now accepts binary data as well and will load the data from `filename` at creation if the `data` parameter is not given.
- Web browser (Selenium):
  - Time between each guild join is now 45 seconds.
  - Selenium can now be used though remote, however it is not recommended.
  - Querying for new guilds will not repeat once no more guilds are found.

v2.9.7

- Fixed channels not being visible though GUI, when using SQL logging.

v2.9.6

- Fixed crash if `start_period` is larger than `end_period`.
- Fixed local update not showing errors if updating objects under AutoGUILD

v2.9.5

- Fixed incorrect caching of the SQL logs, causing incorrect values to be returned back to the GUI.
- Fixed detection of browser automation on searching for new guilds to join.

v2.9.4

- Fixed `AutoGUILD` concurrent access. When updating AutoGUILD, the update method did not block causing exceptions.
- Chrome driver fixes regarding to proxies and timeouts.

v2.9.3

- Fixed `AutoGUILD` and `AutoCHANNEL` regex patterns. Users can now seperate names with "name1 | name2", instead of "name1|name2". #380

v2.9.2

- Fixed viewing dictionaries inside the GUI
- Other bug fixes present in *v2.8.5*

v2.9.1

- Security update for yt-dlp

v2.9

- GUI:
  - Template backups for each structured objects.
  - Rearanging of list items inside GUI listboxes
  - Connection timeout to a remote core is now 10 minutes for large datasets.
  - Dictionary editing - GUI nows allows to edit / view dictionary types (JSON). This could eg. be used to view SQL log's content which is saved to the database into JSON format.
  - Deprecation notices when creating a new object.

  - When opening color chooser and datetime select, the window now opens next to the button instead of window.

- Deprecation:
  - Deprecated Youtube streaming in `AUDIO` in favor of faster loading times. (Scheduled for removal in v2.10)
- Logging:
  - SQL logs can now be deleted though the *`delete_logs()`*.
- Web (browser) layer:
  - Time between guild joins increased to 25 seconds to prevent rate limits.
  - Searching for invite links will be ignored if the user is already joined into the belonging guild.

v2.8.5

- Fixed “Object not added to DAF” when accessing broken accounts from remote

v2.8.4

- Fixed web browser waiting time being too little when searching invite links
- Fixed web browser could not create directory (username had a new line after it, now it auto strips that)
- Fix GUI not allowing to define inherited classes (eg. logging manager’s fallback that inherits LoggerBASE)
- Fix item not in list error upon saving if an item was written inside a GUI’s dropdown menu directly and then edited.

v2.8.3

- Fixed new guilds being added whenever *`daf.client.ACCOUNT`*’s update method failed.
- Fixed error if passing `None` inside update method of account for the `servers` parameter.
- Removed unneded check in object serialization (for remote) which slightly increases performance.
- Fixed Enum values being converted to objects when viewing live items / importing schema from live view.

v2.8.2

- Fixed auto installation of ttkboostrap not opening the main window at the end.

v2.8.1

- Fixed bug `timezone required argument 'offset' when trying to save TextMESSAGE` #325
- Fixed bug `AutoGUILD incorrect type hints` #326

v2.8

- Remote control though HTTP access:
  - The core can be started on a remote server and then connected to and controlled by the graphical interface.
  - The GUI now has a dropdown menu where users can select between a local connection client and a remote connection client. Local connection client won't use the HTTP API, but will start DAF locally and interact with it directly.
- GUI:
  - Method execution
  - Executing method status window.
  - When editing objects, the Y size will now be set to default size every time the frame changes.
  - When executing async blocking functions, a progress bar window will be shown to indicate something is happening.
- Logging:
  - `daf.logging.LoggerJSON` will create a new file once the current one reaches 100 kilobytes.
  - Improved performance of `daf.logging.LoggerJSON`.
  - Loggers will now trace their output path, so users can find the output logs more easily.
- State preservation
  - When using the state preservation (introduced in *v2.7*), accounts that fail to login will, from now on, not be removed from list to prevent data loss.

v2.7

- Preserve objects state on shutdown (accounts, guilds, . . . ,) [logger not preserved]:
  - `daf.core.run()` function's `save_to_file` parameter or *Preserve state on shutdown* checkbox inside *Schema definition* tab of the GUI to configure.
- Analytics:
  - Invite link tracking
  - `GUILD`: `invite_track` parameter for tracking invite links
- File outputs:
  - Changed all paths' defaults to be stored under /<user-home-dir>/daf/ folder to prevent permission problems
- `AutoGUILD` `interval` default changed to `timedelta(minutes=1)`
- xMESSAGE `start_in` now accepts `datetime.datetime` - send at specific datetime.
- GUI:
  - Live object view for viewing and live updating objects.
  - Invite link analytics
  - `Intents` can now also be defined from the GUI.
  - Fixed schema save for enums (enums are not JSON serializable)
- Lowered logging-in timeout to 15 seconds

- **[Breaking change]** Removed DEPRECATED parameters for *daf.core.run()* and *daf.core.initialize()*:
  - token
  - server_list
  - is_user
  - server_log_output
  - sql_manager
  - intents
  - proxy
- **[Breaking change]** Removed DEPRECATED function client.get_client. This is replaced with *daf.core.get_accounts()*, from which the Discord client can be obtained by *daf.client.ACCOUNT.client* for each account.
- **[Breaking change]** Parameter debug in function *daf.core.run()* / *daf.core.initialize()* no longer accepts bool. This was deprecated in some older version and now removed.
- **[Breaking change]** Removed DEPRECATED functionality inside add_object that allowed guilds to be added without passing the account to snowflake parameter. Before it implicitly took the first account from the shill list. This has been deprecated since *v2.4*.
- **[Breaking change]** Removed DEPRECATED functionality inside add_object that allowed snowflake ID and Discord's objects to be passed as snowflake parameter.
- **[Breaking change]** Removed DEPRECATED function get_guild_user, which has been deprecated since *v2.4*.
- **[Breaking change]** xMESSAGE types no longer accept bool for parameter start_in. This has been deprecated since *v2.1*.

v2.6.3

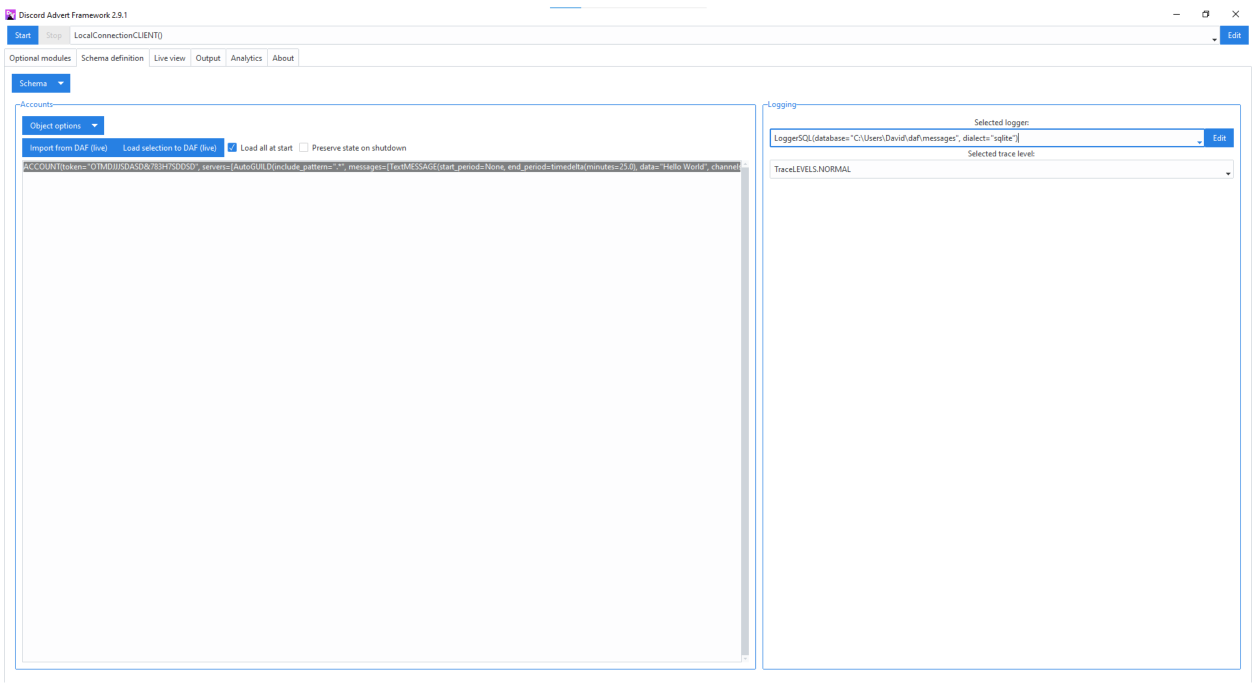
- Restored support for Python v3.8

v2.6.1

- Fixed logger not being converted properly when exporting GUI data into a script.

v2.6.0

- Graphical User Interface - **GUI** for controlling the framework, defining the schema (with backup and restore) and script generation!



LocalConnectionCLIENT()

✓ Load all at start

Preserve state on shutdown

```
ACCOUNT(token="OTMDJJJSDASD&783H7SDDSD", servers=[AutoGUILD(include_pattern=".*", messages=[TextMESSAGE(start_period=None, end_period=timedelta(minutes=25.0), data="Hello World", channel
```

Selected logger: LoggerSQL(database="C:\Users\David\daf\messages", dialect="sqlite")

Selected trace level: TraceLEVELS.NORMAL

- Logging:
  - Added `author` field to all logging managers (tells us which account sent the message).
  - SQL analysis

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:**

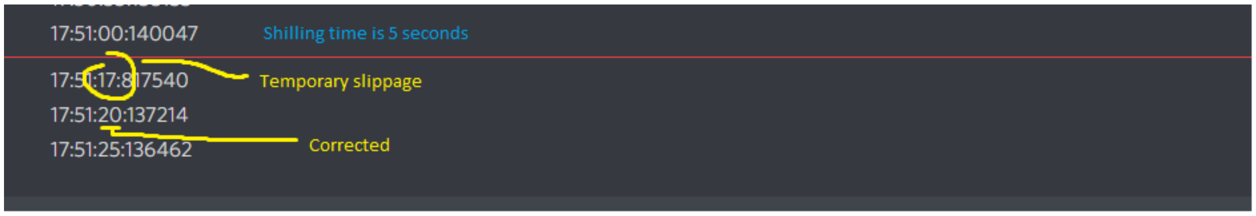
  `daf.TextMESSAGE` and `daf.VoiceMESSAGE` now check if the given channels are actually inside the guild
- **Optionals:**

  [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.



```
17:51:00:140047
17:51:17:817540
17:51:20:137214
17:51:25:136462
```

Fig. 5.5: 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