dislash.py

Release 1.4.7

EQUENOS

Aug 20, 2021

CONTENTS

This library adds different interaction features to discord.py, such as buttons or slash commands.

It's recommended to read *Quickstart* first.

CHAPTER

ONE

SECTIONS

1.1 Quickstart

1.1.1 Installation

Enter one of these commands to install the library:

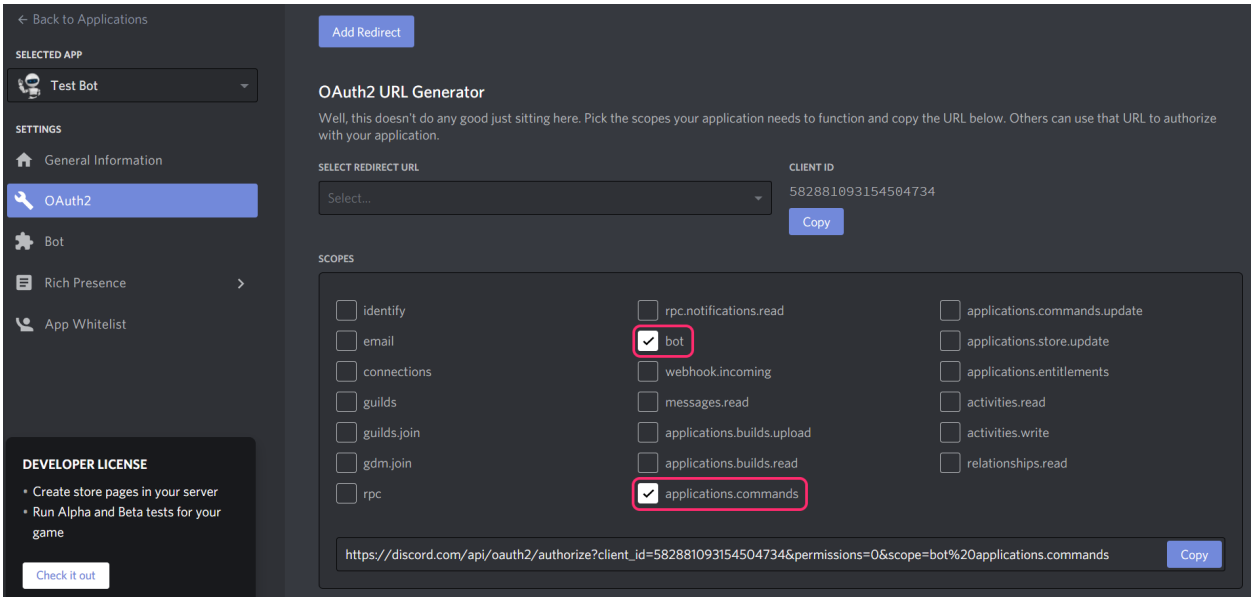
```
pip install dislash.py
```

```
python -m pip install dislash.py
```

Or just clone the repo: https://github.com/EQUENOS/dislash.py

1.1.2 Authorising

Before we start, make sure your bot has `application.commands` scope, in case you want to make some slash commands.
In order to grant it, authorise (or reauthorise) your bot with this tick pressed:



1.1.3 Creating a simple command

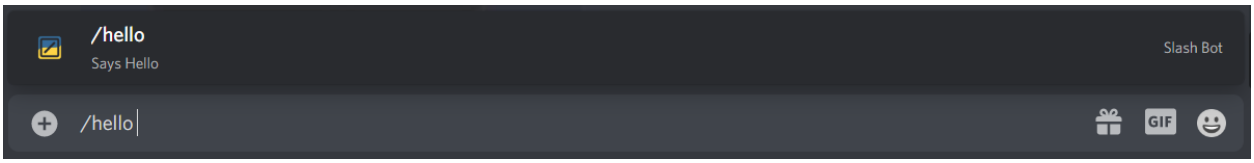
Let's make a **/hello** command that will send "Hello!" to the chat.

```
from discord.ext import commands
from dislash import InteractionClient

bot = commands.Bot(command_prefix="!")
# test_guilds param is optional, this is a list of guild IDs
inter_client = InteractionClient(bot, test_guilds=[12345])

@inter_client.slash_command(description="Says Hello")
async def hello(ctx):
    await ctx.send("Hello!")

bot.run("BOT_TOKEN")
```

Note:

Per-guild registration is **instant**, while global registration takes up to **1 hour** to complete.
In order to register a command globally, do not specify the `test_guilds` / `guild_ids` parameters.

And here we go! We've just made a simple slash-command named **/hello**



1.1.4 Playing with buttons

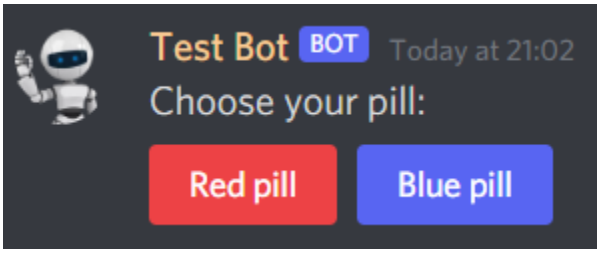
Let's make a text command that sends 2 buttons and removes them as soon as one of them is pressed.

```
from discord.ext import commands
from dislash import InteractionClient, ActionRow, Button, ButtonStyle

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

@bot.command()
async def test(ctx):
    # Create a row of buttons
    row = ActionRow(
        Button(
            style=ButtonStyle.red,
            label="Red pill",
            custom_id="red_pill"
        ),
        Button(
            style=ButtonStyle.blurple,
            label="Blue pill",
```

(continues on next page)

(continued from previous page)

```
            custom_id="blue_pill"
        )
    )
    # Note that we assign a list of rows to components
    msg = await ctx.send("Choose your pill:", components=[row])
    # This is the check for button_click waiter
    def check(inter):
        return inter.author == ctx.author
    # Wait for a button click under the bot's message
    inter = await msg.wait_for_button_click(check=check)
    # Respond to the interaction
    await inter.reply(
        f"Your choice: {inter.clicked_button.label}",
        components=[] # This is how you remove buttons
    )

bot.run("BOT_TOKEN")
```



Note: *InteractionClient* should always be called in the main file, even if you're not making any slash commands. This class modifies some **discord.py** methods so they work with buttons.

1.1.5 More examples

Note: For more examples, see *Examples*

1.2 Examples

1.2.1 A simple slash command

```
from discord.ext import commands
from dislash import InteractionClient

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

@inter_client.slash_command(description="Sends Hello")
async def hello(interaction):
```

(continues on next page)

(continued from previous page)

```
    await interaction.reply("Hello!")

bot.run("BOT_TOKEN")
```

See also:

What's interaction? See *SlashInteraction* to learn more.

1.2.2 Slash embed

Let's make something more complicated than **/hello**. For example, a command that generates an embed.

```
import discord
from discord.ext import commands
from dislash import InteractionClient, Option, OptionType

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot)
test_guilds = [12345]   # Insert ID of your guild here

@inter_client.slash_command(
    guild_ids=test_guilds,
    description="Builds a custom embed",
    options=[
        Option('title', 'Makes the title of the embed', OptionType.STRING),
        Option('description', 'Makes the description', OptionType.STRING),
        Option('color', 'The color of the embed', OptionType.STRING)

        # Note that all args are optional
        # because we didn't specify required=True in Options
    ]
)
async def embed(inter, title=None, description=None, color=None):
    # Converting color
    if color is not None:
        try:
            color = await commands.ColorConverter().convert(inter, color)
        except:
            color = None
    if color is None:
        color = discord.Color.default()
    # Generating an embed
    emb = discord.Embed(color=color)
    if title is not None:
        emb.title = title
    if desc is not None:
        emb.description = desc
    # Sending the output
    await inter.respond(embed=emb, hide_user_input=True)
```

(continues on next page)

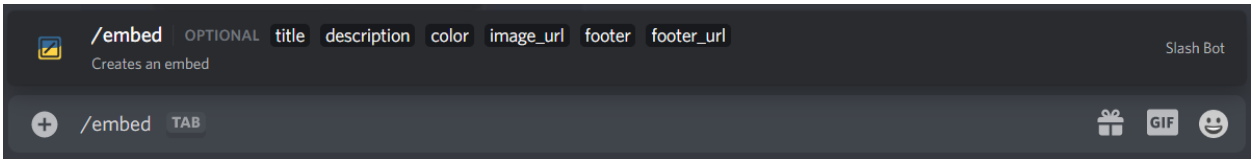
(continued from previous page)

```
bot.run("BOT_TOKEN")
```

See also:

Option to learn more about slash command options.

Here's the result we've just achieved:

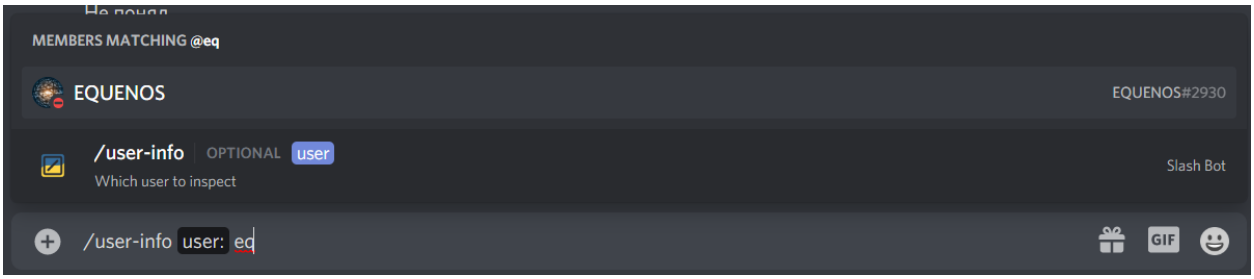


1.2.3 Slash user-info

It's time to work with different argument types. This example shows how to easily make a **/user-info** command

```
import discord
from discord.ext import commands
from dislash import InteractionClient, Option, OptionType

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot)
test_guilds = [12345]

@inter_client.slash_command(
    guild_ids=test_guilds,
    name="user-info",
    description="Shows user's profile",
    options=[
        Option("user", "Specify any user", OptionType.USER),
    ]
)
async def user_info(inter, user=None):
    # Default user is the command author
    user = user or inter.author

    emb = discord.Embed(color=discord.Color.blurple())
    emb.title = str(user)
    emb.description = (
        f"**Created at:** `{user.created_at}`\n"
        f"**ID:** `{user.id}`"
    )
    emb.set_thumbnail(url=user.avatar_url)
    await inter.respond(embed=emb)

bot.run("BOT_TOKEN")
```

Here's how this slash command looks like in Discord:



1.2.4 Buttons

```
from discord.ext import commands
from dislash import InteractionClient, ActionRow, Button, ButtonStyle

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

@bot.command()
async def test(ctx):
    # Make a row of buttons
    row_of_buttons = ActionRow(
        Button(
            style=ButtonStyle.green,
            label="Green button",
            custom_id="green"
        ),
        Button(
            style=ButtonStyle.red,
            label="Red button",
            custom_id="red"
        )
    )
    # Send a message with buttons
    msg = await ctx.send(
        "This message has buttons!",
        components=[row_of_buttons]
    )
    # Wait for someone to click on them
    inter = await msg.wait_for_button_click(check)
    # Send what you received
    button_text = inter.clicked_button.label
    await inter.reply(f"Button: {button_text}")

bot.run("BOT_TOKEN")
```

1.2.5 Context menus

This example shows how to create context menu commands and interact with them. Context menu commands are actions that can be triggered from user and message context menus.

```
from discord.ext import commands
from dislash import InteractionClient

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

@inter_client.user_command(name="Press me")
async def press_me(inter):
    # User commands are visible in user context menus
    # They can be global or per guild, just like slash commands
    await inter.respond(f"Hello {inter.author} and {inter.target}")

@inter_client.message_command(name="Resend")
async def resend(inter):
    # Message commands are visible in message context menus
    # inter is instance of ContextMenuInteraction
    await inter.respond(inter.message.content)

bot.run("BOT_TOKEN")
```

1.3 Buttons

1.3.1 Basic example

Let's make a simple command that waits for button clicks and deletes the button on timeout.

In this example we're using the following objects and methods:

- *InteractionClient* to enable the extension
- *ActionRow* to make a row of buttons
- *Button* to design the buttons
- *ClickListener* to process the button clicks
- – *ClickListener.matching_id*
- – *ClickListener.timeout*

```
from discord.ext import commands
from dislash import InteractionClient, ActionRow, Button, ButtonStyle

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

@bot.command()
async def test(ctx):
    row = ActionRow(
        Button(
```

(continues on next page)

(continued from previous page)

```
            style=ButtonStyle.green,
            label="Click me!",
            custom_id="test_button"
        )
    )
    msg = await ctx.send("I have a button!", components=[row])

    # Here timeout=60 means that the listener will
    # finish working after 60 seconds of inactivity
    on_click = msg.create_click_listener(timeout=60)

    @on_click.matching_id("test_button")
    async def on_test_button(inter):
        await inter.reply("You've clicked the button!")

    @on_click.timeout
    async def on_timeout():
        await msg.edit(components=[])

bot.run("BOT_TOKEN")
```

Note: All decorated functions, except for the timeout function, must take **only one** argument, which is guaranteed to be an instance of *MessageInteraction*.

1.3.2 Adding layers

Let's say we don't want any other person except for the command author to click the buttons. It's time to work with *ClickListener.not_from_user* decorator.

```
from discord.ext import commands
from dislash import InteractionClient, ActionRow, Button, ButtonStyle

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

@bot.command()
async def test(ctx):
    row = ActionRow(
        Button(
            style=ButtonStyle.green,
            label="Click me!",
            custom_id="test_button"
        )
    )
    msg = await ctx.send("I have a button!", components=[row])

    # Here timeout=60 means that the listener will
    # finish working after 60 seconds of inactivity
    on_click = msg.create_click_listener(timeout=60)
```

(continues on next page)

(continued from previous page)

```
    @on_click.not_from_user(ctx.author, cancel_others=True, reset_timeout=False)
    async def on_wrong_user(inter):
        # This function is called in case a button was clicked not by the author
        # cancel_others=True prevents all on_click-functions under this function from␣
↪working
        # regardless of their checks
        # reset_timeout=False makes the timer keep going after this function is called
        await inter.reply("You're not the author", ephemeral=True)

    @on_click.matching_id("test_button")
    async def on_test_button(inter):
        # This function only works if the author presses the button
        # Becase otherwise the previous decorator cancels this one
        await inter.reply("You've clicked the button!")

    @on_click.timeout
    async def on_timeout():
        await msg.edit(components=[])

bot.run("BOT_TOKEN")
```

Note: The check must take **only one** argument, which is guaranteed to be an instance of *MessageInteraction*. It also must return a `boolean` value.

The bot is now respoding to all strangers with a hidden message and prevents them from clicking the buttons. Note that we specified `cancel_others=True`. This means that the click manager won't toggle other `@on_click...` listeners if the author-check was activated.

What's more, the click manager doesn't reset the timeout if a stranger clicks the button. In other words, even if the command author is no longer pressing the buttons, other users won't prevent the click listener from exceeding the timeout. We achieved this by setting the `reset_timeout` paramter to `False`.

1.4 Select Menus

1.4.1 Sending a menu

Let's make a simple command that sends a menu.

In this example we're using the following objects and methods:

- *InteractionClient* to enable the extension
- *SelectMenu* to build the menu

```
from discord.ext import commands
from dislash import InteractionClient, SelectMenu

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

(continues on next page)

(continued from previous page)

```
@bot.command()
async def test(ctx):
    await ctx.send(
        "This message has a select menu!",
        components=[
            SelectMenu(
                custom_id="test",
                placeholder="Choose up to 2 options",
                max_values=2,
                options=[
                    SelectOption("Option 1", "value 1"),
                    SelectOption("Option 2", "value 2"),
                    SelectOption("Option 3", "value 3")
                ]
            )
        ]
    )

bot.run("BOT_TOKEN")
```

1.4.2 Responding to a menu click

Let's send a menu and then respond to the first click.

```
from discord.ext import commands
from dislash import InteractionClient, SelectMenu

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

@bot.command()
async def test(ctx):
    msg = await ctx.send(
        "This message has a select menu!",
        components=[
            SelectMenu(
                custom_id="test",
                placeholder="Choose up to 2 options",
                max_values=2,
                options=[
                    SelectOption("Option 1", "value 1"),
                    SelectOption("Option 2", "value 2"),
                    SelectOption("Option 3", "value 3")
                ]
            )
        ]
    )
    def check(inter):
        # inter is instance of MessageInteraction
        # read more about it in "Objects and methods" section
        if inter.author == ctx.author
```

(continues on next page)

(continued from previous page)

```
    # Wait for a menu click under the message you've just sent
    inter = await msg.wait_for_dropdown(check)
    # Tell which options you received
    labels = [option.label for option in inter.select_menu.selected_options]
    await inter.reply(f"Your choices: {', '.join(labels)}")

bot.run("BOT_TOKEN")
```

Here we used `Message.wait_for_dropdown` method to receive an interaction with the menu. This is cool, but if you want to track menu interactions permanently, try using the `on_dropdown` event.

```
@bot.event
async def on_dropdown(inter: MessageInteraction):
    # ...
```

1.5 Slash commands

1.5.1 Introduction

As you've already noticed, Discord makes all slash commands look like a part of the interface. This is possible because every slash command is registered before people can use it. Even though this library registers your commands automatically, you should still design every slash command yourself ;)

What are interactions?

1. User inputs data using special interface
2. Discord converts this data into valid command args
3. Discord API sends the data to your app

The data you receive is called *SlashInteraction*.

There're 2 types of slash commands: global and local (per guild). Global commands are visible everywhere, including bot DMs. Per guild commands are only visible in corresponding guilds.

Note:

Global command registration takes more than **1 hour**.
Guild command registration is **instant**.

1.5.2 Basic example

In this example we're using the following objects and methods:

- *InteractionClient* to activate the extension
- `SlashClient.command` to make a command
- *SlashInteraction* represented by `inter` (see the code below)

```
from discord.ext import commands
from dislash import InteractionClient

bot = commands.Bot(command_prefix="!")
# test_guilds param is an optional list of guild IDs
inter_client = InteractionClient(bot, test_guilds=[12345])

@inter_client.slash_command(description="Test command")
async def test(inter):
    await inter.reply("Test")

bot.run("BOT_TOKEN")
```

1.5.3 A command with arguments

Let's make a command that shows the avatar of the user. If user isn't specified, it shows the avatar of the author.

In addition to all previous methods, we're going to use these:

- *Option* to make an option
- *OptionType* to specify the option type

This is required for further command registration.

```
import discord
from discord.ext import commands
from dislash import InteractionClient, Option, OptionType

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot, test_guilds=[12345])

@inter_client.slash_command(
    description="Shows the avatar of the user",
    options=[
        Option("user", "Enter the user", OptionType.USER)
        # By default, Option is optional
        # Pass required=True to make it a required arg
    ]
)
async def avatar(inter, user=None):
    # If user is None, set it to inter.author
    user = user or inter.author
    # We are guaranteed to receive a discord.User object,
    # because we specified the option type as Type.USER
```

(continues on next page)

(continued from previous page)

```
    emb = discord.Embed(
        title=f"{user}'s avatar",
        color=discord.Color.blue()
    )
    emb.set_image(url=user.avatar_url)
    await inter.reply(embed=emb)

bot.run("BOT_TOKEN")
```

1.5.4 Slash subcommands

Creating subcommands is as easy as creating commands. The only difference is the decorator we use.

In addition to all previous methods, we're going to use *CommandParent.sub_command* (represented by `say.sub_command` in the code)

```
from discord.ext import commands
from dislash import InteractionClient

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot, test_guilds=[12345])

@inter_client.slash_command(description="Has subcommands")
async def say(inter):
    # This is just a parent for 2 subcommands
    # It's not necessary to do anything here,
    # but if you do, it runs for any subcommand nested below
    pass

# For each subcommand you can specify individual options and other parameters,
# see the "Objects and methods" reference to learn more.
@say.sub_command(description="Says hello")
async def hello(inter):
    await inter.reply("Hello!")

@say.sub_command(description="Says bye")
async def goodbye(inter):
    await inter.reply("Bye!")

bot.run("BOT_TOKEN")
```

1.5.5 Subcommand groups

You can make a command with groups of subcommands using *CommandParent.sub_command_group* and *SubCommandGroup.sub_command*

Partial code:

```
@inter_client.slash_command(description="Has groups")
async def groups(inter):
    pass
```

(continues on next page)

(continued from previous page)

```
@groups.sub_command_group()
async def group_1(inter):
    pass

@group_1.sub_command()
async def blue(inter):
    # This will be displayed as
    # /groups group_1 blue
    pass

@group_1.sub_command()
async def green(inter):
    # This will be displayed as
    # /groups group_1 green
    pass

@groups.sub_command_group()
async def group_2(inter):
    # You got the idea
    pass

bot.run("BOT_TOKEN")
```

1.6 Context Menus

1.6.1 Introduction

There're 2 types of context menus:

- User commands
- Message commands

If you right click on a user and hover over the "Apps" section you'll see the list of user commands. If there's no section named "Apps", it means that there're no user commands yet.

In order to find the list of message commands do the same actions with a message. Hover over the "Apps" section and that's it.

Context menu in Discord API is actually a sub section of application commands, just like slash commands. This is why creating a context menu is really similar to creating a slash command.

1.6.2 Making a user command

In this example we're using the following objects and methods:

- *InteractionClient* to enable the extension
- *ContextMenuInteraction* for a typehint

```
from discord.ext import commands
from dislash import InteractionClient, ContextMenuInteraction

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot, test_guilds=[12345])
# test_guilds is a list of guilds for testing your application commands.
# The list of app commands will only be visible there.
# In order to make the commands globally visible, don't specify the test_guilds.

@inter_client.user_command(name="Created at")
async def created_at(inter: ContextMenuInteraction):
    # User commands always have only this ^ argument
    await inter.respond(
        f"{inter.user} was created at {inter.user.created_at}",
        ephemeral=True # Make the message visible only to the author
    )

bot.run("BOT_TOKEN")
```

1.6.3 Making a message command

In this example we're using the same objects and methods.

```
from discord.ext import commands
from dislash import InteractionClient, ContextMenuInteraction

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot, test_guilds=[12345])
# test_guilds is a list of guilds for testing your application commands.
# The list of app commands will only be visible there.
# In order to make the commands globally visible, don't specify the test_guilds.

@inter_client.message_command(name="Reverse")
async def reverse(inter: ContextMenuInteraction):
    # Message commands always have only this ^ argument
    if inter.message.content:
        # Here we will send a reversed message to the chat
        await inter.respond(inter.message.content[::-1])
    else:
        # Here we will explain that the message isn't valid
        await inter.respond("There's no content", ephemeral=True)

bot.run("BOT_TOKEN")
```

1.6.4 Handling errors

You can make local and global error handlers, which are similar to discord.py error handlers.

Example of a global error handler:

```
from discord.ext import commands
from dislash import InteractionClient, ContextMenuInteraction

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot, test_guilds=[12345])

@inter_client.user_command(name="Created at")
async def created_at(inter: ContextMenuInteraction):
    await inter.respond(
        f"{inter.user} was created at {inter.user.created_at}",
        ephemeral=True
    )

@bot.event
async def on_user_command_error(inter: ContextMenuInteraction, error):
    # This is a global error handler for user commands.
    await inter.respond(f"Failed to execute {inter.user_command.name} due to {error}")

bot.run("BOT_TOKEN")
```

Example of a local error handler:

```
from discord.ext import commands
from dislash import InteractionClient, ContextMenuInteraction

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot, test_guilds=[12345])

@inter_client.message_command(name="Reverse")
async def reverse(inter: ContextMenuInteraction):
    await inter.respond(inter.message.content[::-1])

@reverse.error
async def on_reverse_error(inter: ContextMenuInteraction, error):
    # This is a local error handler specifically for "reverse" command
    await inter.respond("This message is invalid", ephemeral=True)

bot.run("BOT_TOKEN")
```

1.6.5 Adding checks and cooldowns

You can add some checks to an application command similarly to usual commands from discord.py. All checks from the *Checks* section work for any kind of application commands.

```
from discord.ext import commands
from dislash import InteractionClient, ContextMenuInteraction, application_commands

bot = commands.Bot(command_prefix="!")
inter_client = InteractionClient(bot, test_guilds=[12345])

# Here we set the command cooldown as 5 seconds per command per user
@application_commands.cooldown(1, 5, application_commands.BucketType.user)
@inter_client.message_command(name="Reverse")
async def reverse(inter: ContextMenuInteraction):
    await inter.respond(inter.message.content[::-1])

@bot.event
async def on_message_command_error(inter: ContextMenuInteraction, error):
    if isinstance(error, application_commands.CommandOnCooldown):
        await inter.respond(f"Try again in {error.retry_after:.1f} s", ephemeral=True)

bot.run("BOT_TOKEN")
```

1.6.6 Relevant events

```
@bot.event
async def on_user_command(inter: ContextMenuInteraction):
    # Toggles if someone interacts with a user command of your app
    ...
```

```
@bot.event
async def on_message_command(inter: ContextMenuInteraction):
    # Toggles if someone interacts with a message command of your app
    ...
```

1.7 Cogs

Sorting commands between cogs is a popular practice in bot development. This section shows how to build slash commands in cogs.

It's as simple as this:

1.7.1 Example

```
from discord.ext import commands
from dislash import slash_command, ActionRow, Button, ButtonStyle


class mycog(commands.Cog):
    def __init__(self, bot):
        self.bot = bot

    # Example of a slash command in a cog
    @slash_command(description="Says Hello")
    async def hello(self, inter):
        await inter.respond("Hello from cog!")

    # Buttons in cogs (no changes basically)
    @commands.command()
    async def test(self, ctx):
        row_of_buttons = ActionRow(
            Button(
                style=ButtonStyle.green,
                label="Green button",
                custom_id="green"
            ),
            Button(
                style=ButtonStyle.red,
                label="Red button",
                custom_id="red"
            )
        )
        msg = await ctx.send("This message has buttons", components=[row_of_buttons])
        # Wait for a button click
        def check(inter):
            return inter.author == ctx.author
        inter = await msg.wait_for_button_click(check=check)
        # Process the button click
        inter.reply(f"Button: {inter.button.label}", type=ResponseType.UpdateMessage)

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

1.7.2 Differences

- `@slash_command` instead of `@InteractionClient.slash_command`
- `self` is a required first argument
- `self.bot` instead of `bot`
- `InteractionClient` is accessible via `self.bot.slash`

1.8 Events

The library provides you with some new events related to interactions.
Let's assume that you have this in your code:

```
from discord.ext import commands
from dislash import *

bot = commands.Bot(command_prefix="!")
slash = SlashClient(bot)
```

Here're 3 different ways of working with dislash events:

```
@slash.event
async def on_event():
    # ...
```

```
@bot.listen()
async def on_event():
    # ...
```

```
# For cogs
@commands.Cog.listener()
async def on_event(self):
    # ...
```

1.8.1 on_ready

async events.**on_ready**()
An event which is activated when the dislash extension is ready and all slash commands are synced (if there're any)

1.8.2 on_auto_register

async events.**on_auto_register**(*global_commands_patched*, *patched_guilds*)
An event which is called after auto synchronisation of commands.

Parameters

- **global_commands_patched** (bool) – whether the global application commands were updated
- **patched_guilds** (List[int]) – the list of IDs of guilds where the commands were updated

1.8.3 on_button_click

async events.**on_button_click**(*interaction*)
An event which is called on any button click of your application.

Parameters **interaction** (MessageInteraction) – the interaction with the button

1.8.4 on_dropdown

async events.**on_dropdown**(*interaction*)
An event which is called on any menu click of your application.

Parameters **interaction** (MessageInteraction) – the interaction with the select menu

1.8.5 on_slash_command

async events.**on_slash_command**(*interaction*)
An event which is called every time a slash command of your application is invoked.

Parameters **interaction** (SlashInteraction) – the interaction with a slash command

1.8.6 on_user_command

async events.**on_user_command**(*interaction*)
An event which is called every time a user command of your application is invoked.

Parameters **interaction** (ContextMenuInteraction) – the interaction with a user command

1.8.7 on_message_command

async events.**on_message_command**(*interaction*)
An event which is called every time a message command of your application is invoked.

Parameters **interaction** (ContextMenuInteraction) – the interaction with a message command

1.8.8 on_slash_command_error

async events.**on_slash_command_error**(*interaction*, *error*)

An event which is called every time a slash command fails due to some error. This also includes InteractionCheckFailure

Parameters

- **interaction** (SlashInteraction) – the interaction with a slash command
- **error** (ApplicationCommandError) – the error that was raised

1.8.9 on_user_command_error

async events.**on_user_command_error**(*interaction*, *error*)

An event which is called every time a user command fails due to some error. This also includes InteractionCheckFailure

Parameters

- **interaction** (ContextMenuInteraction) – the interaction with a user command
- **error** (ApplicationCommandError) – the error that was raised

1.8.10 on_message_command_error

async events.**on_message_command_error**(*interaction*, *error*)

An event which is called every time a message command fails due to some error. This also includes InteractionCheckFailure

Parameters

- **interaction** (ContextMenuInteraction) – the interaction with a message command
- **error** (ApplicationCommandError) – the error that was raised

1.9 Slash command checks

1.9.1 Introduction

This section contains documentation of decorator-based checks that **dislash.py** has.

Here's a possible use case:

```
import dislash
from discord.ext import commands

bot = commands.Bot(command_prefix="!")
inter_client = dislash.InteractionClient(bot)

@inter_client.slash_command(description="A command for admins")
@dislash.has_permissions(administrator=True)
async def hello(inter):
    await inter.reply("Hello, administrator!")
```

(continues on next page)

(continued from previous page)

```
bot.run("BOT_TOKEN")
```

If you want to respond with something like *"You're missing permissions!"*, you should add an error handler using `on_slash_command_error(inter, error)` event, learn more about it here: *on_slash_command_error*

1.9.2 Checks

dislash.**check**(*predicate*)

A function that converts `predicate(interaction)` functions into application command decorators

Example

```
def is_guild_owner():
    def predicate(inter):
        return inter.author.id == inter.guild.owner_id
    return check(predicate)

@is_guild_owner()
@slash.command(description="Says Hello if you own the guild")
async def hello(inter):
    await inter.reply("Hello, Mr.Owner!")
```

dislash.**check_any**(**checks*)

Similar to `commands.check_any`

dislash.**cooldown**(*rate*, *per*, *type=<BucketType.default: 0>*)

A decorator that adds a cooldown to a slash-command. Similar to **discord.py** cooldown decorator.

A cooldown allows a command to only be used a specific amount of times in a specific time frame. These cooldowns can be based either on a per-guild, per-channel, per-user, per-role or global basis. Denoted by the third argument of `type` which must be of enum type `BucketType`.

If a cooldown is triggered, then `CommandOnCooldown` is triggered in `on_slash_command_error` in the local error handler.

A command can only have a single cooldown.

Parameters

- **rate** (*int*) – The number of times a command can be used before triggering a cooldown.
- **per** (*float*) – The amount of seconds to wait for a cooldown when it's been triggered.
- **type** (BucketType) – The type of cooldown to have.

dislash.**has_role**(*item*)

Similar to `commands.has_role`

dislash.**has_any_role**(**items*)

Similar to `commands.has_any_role`

dislash.**has_permissions**(***perms*)

Similar to `commands.has_permissions`

dislash.**has_guild_permissions**(***perms*)

Similar to `commands.has_guild_permissions`

dislash.**dm_only**()
Similar to commands.dm_only

dislash.**guild_only**()
Similar to commands.guild_only

dislash.**is_owner**()
Similar to commands.is_owner

dislash.**is_nsfw**()
Similar to commands.is_nsfw

dislash.**bot_has_role**(*item*)
Similar to commands.bot_has_role

dislash.**bot_has_any_role**(**items*)
Similar to commands.bot_has_any_role

dislash.**bot_has_permissions**(***perms*)
Similar to commands.bot_has_permissions

dislash.**bot_has_guild_permissions**(***perms*)
Similar to commands.bot_has_guild_permissions

1.10 Objects and methods

1.10.1 ActionRow

class dislash.**ActionRow**(**components*)
Represents an action row. Action rows are basically shelves for buttons.

Parameters **components** (List[Button]) – a list of up to 5 buttons to place in a row

add_button(*, *style:* dislash.interactions.message_components.ButtonStyle, *label: Optional[str] = None*, *emoji: Optional[str] = None*, *custom_id: Optional[str] = None*, *url: Optional[str] = None*, *disabled: bool = False*)

add_menu(*, *custom_id: str*, *placeholder: Optional[str] = None*, *min_values: int = 1*, *max_values: int = 1*, *options: Optional[list] = None*)

disable_buttons(**positions: int*)
Sets disabled to True for all buttons in this row.

enable_buttons(**positions: int*)
Sets disabled to False for all buttons in this row.

classmethod **from_dict**(*data: dict*)

to_dict()

1.10.2 ApplicationCommand

class dislash.**ApplicationCommand**(*type:* dislash.interactions.application_command.ApplicationCommandType, ***kwargs*)

Base class for application commands

1.10.3 ApplicationCommandError

class dislash.**ApplicationCommandError**(*message=None*, **args*)

The base exception type for all slash-command related errors.

This inherits from `discord.DiscordException`.

This exception and exceptions inherited from it are handled in a special way as they are caught and passed into a special event from `SlashClient`, `on_slash_command_error()`.

1.10.4 ApplicationCommandInteractionData

class dislash.**ApplicationCommandInteractionData**(*, *data*, *guild*, *state*)

1.10.5 ApplicationCommandPermissions

class dislash.**ApplicationCommandPermissions**(*raw_permissions: Optional[list] = None*)

Represents application command permissions. Roughly equivalent to a list of *RawCommandPermission*

Application command permissions are checked on the server side. Only local application commands can have this type of permissions.

Parameters **raw_permissions** (*List[RawCommandPermission]*) – a list of *RawCommandPermission*. However, *from_pairs()* or *from_ids()* might be more convenient.

classmethod from_dict(*data: dict*)

classmethod from_ids(*role_perms: Optional[dict] = None*, *user_perms: Optional[dict] = None*)

Creates `SlashCommandPermissions` from 2 dictionaries of IDs and permissions.

Parameters

- **role_perms** (`dict`) – a dictionary of {`role_id`: `bool`}
- **user_perms** (`dict`) – a dictionary of {`user_id`: `bool`}

classmethod from_pairs(*permissions: dict*)

Creates `SlashCommandPermissions` using instances of `discord.Role` and `discord.User`

Parameters **permissions** (`dict`) – a dictionary of {`Role | User`: `bool`}

to_dict()

1.10.6 ApplicationCommandType

class dislash.**ApplicationCommandType**

1.10.7 BadArgument

class dislash.**BadArgument**(*message=None*, **args*)

1.10.8 BaseInteraction

class dislash.**BaseInteraction**(*client*, *data: dict*)

The base class for all interactions

async **create_response**(*content=None*, ***, *type=None*, *embed=None*, *embeds=None*, *components=None*, *view=None*, *ephemeral=False*, *tts=False*, *allowed_mentions=None*)

Creates an interaction response.

Parameters

- **content** (str) – response content
- **type** (int | *ResponseType*) – sets the response type. See *ResponseType*
- **embed** (discord.Embed) – response embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both embed and embeds are specified

async **delete**()

Deletes the original interaction response.

async **delete_after**(*delay: float*)

async **edit**(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *allowed_mentions=None*)

Edits the original interaction response.

Parameters

- **content** (str) – New message content
- **embed** (discord.Embed) – New message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds of a new message
- **components** (List[ActionRow]) – a list of up to 5 action rows

- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Returns **message** – The message that was edited

Return type discord.Message

async fetch_initial_response()
Fetches the original interaction response.

async followup(*content=None*, ***, *embed=None*, *embeds=None*, *file=None*, *files=None*, *components=None*, *view=None*, *tts=False*, *ephemeral=False*, *allowed_mentions=None*, *username=None*, *avatar_url=None*)
Sends a followup message.

Parameters

- **content** (str) – the followup message content
- **embed** (discord.Embed) – the followup message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **file** (discord.File) – a file to attach to the message
- **files** (List[discord.File]) – a list of files to attach to the message
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your message will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message
- **username** (str) – override the default bot name
- **avatar_url** (str) – override the default avatar of the bot

async reply(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *view=None*, *file=None*, *files=None*, *tts=False*, *hide_user_input=False*, *ephemeral=False*, *delete_after=None*, *allowed_mentions=None*, *type=None*, *fetch_response_message=True*)
Creates an interaction response. What's the difference between this method and *create_response()*? If the token is no longer valid, this method sends a usual channel message instead of creating an interaction response. Also, this method fetches the interaction response message and returns it, unlike *create_response()*.

Parameters

- **content** (str) – message content
- **embed** (discord.Embed) – message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation

- **file** (discord.File) – if it's the first interaction reply, the file will be ignored due to API limitations. Everything else is the same as in discord.TextChannel.send() method.
- **files** (List[discord.File]) – same as file but for multiple files.
- **hide_user_input** (bool) – if set to True, user's input won't be displayed
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **delete_after** (float) – if specified, your reply will be deleted after delete_after seconds
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.
- **type** (int | *ResponseType*) – sets the response type. If it's not specified, this method sets it according to hide_user_input, content and embed params.
- **fetch_response_message** (bool) – whether to fetch and return the response message. Defaults to True.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both embed and embeds are specified

Returns **message** – The response message that has been sent or None if the message is ephemeral

Return type discord.Message | None

1.10.9 BotMissingAnyRole

class dislash.**BotMissingAnyRole**(*missing_roles*)

1.10.10 BotMissingPermissions

class dislash.**BotMissingPermissions**(*missing_perms*, **args*)

1.10.11 BotMissingRole

class dislash.**BotMissingRole**(*missing_role*)

1.10.12 BucketType

class dislash.**BucketType**(*value*)

classmethod try_value(*value*)

1.10.13 Button

class dislash.**Button**(*, *style:* dislash.interactions.message_components.ButtonStyle, *label: Optional[str] = None*, *emoji: Optional[discord.partial_emoji.PartialEmoji] = None*, *custom_id: Optional[str] = None*, *url: Optional[str] = None*, *disabled: bool = False*)

Builds a button.

Parameters

- **style** (*ButtonStyle*) – Style of the button
- **label** (str) – Button text
- **emoji** (str | discord.PartialEmoji) – Button emoji
- **custom_id** (str) – You should set it by yourself, it's not a snowflake. If button style is not ButtonStyle.link, this is a required field
- **url** (str) – If button style is ButtonStyle.link, this is a required field.
- **disabled** (bool) – Whether the button is disabled or not. Defaults to false.

classmethod from_dict(*data: dict*)

to_dict()

1.10.14 MessageInteraction

class dislash.**MessageInteraction**(*client*, *data*)

Represents a button interaction. Obtainable via discord.Context.wait_for_button_click and in on_button_click event.

author

The user that clicked the button

Type discord.Member | discord.User

channel

The channel where the click happened

Type discord.Messageable

guild

The guild where the click happened

Type discord.Guild | None

message

The message where the button was clicked

Type discord.Message

components

A list of *ActionRow* instances containing other components

Type list

component

The component that author interacted with

Type *Component*

async create_response(*content=None*, ***, *type=None*, *embed=None*, *embeds=None*, *components=None*, *view=None*, *ephemeral=False*, *tts=False*, *allowed_mentions=None*)

Creates an interaction response.

Parameters

- **content** (str) – response content
- **type** (int | *ResponseType*) – sets the response type. See *ResponseType*
- **embed** (discord.Embed) – response embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both embed and embeds are specified

async delete()

Deletes the original interaction response.

async delete_after(*delay: float*)

async edit(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *allowed_mentions=None*)

Edits the original interaction response.

Parameters

- **content** (str) – New message content
- **embed** (discord.Embed) – New message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds of a new message
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Returns **message** – The message that was edited

Return type discord.Message

async fetch_initial_response()

Fetches the original interaction response.

async followup(*content=None*, ***, *embed=None*, *embeds=None*, *file=None*, *files=None*, *components=None*, *view=None*, *tts=False*, *ephemeral=False*, *allowed_mentions=None*, *username=None*, *avatar_url=None*)

Sends a followup message.

Parameters

- **content** (str) – the followup message content
- **embed** (discord.Embed) – the followup message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **file** (discord.File) – a file to attach to the message
- **files** (List[discord.File]) – a list of files to attach to the message
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your message will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message
- **username** (str) – override the default bot name
- **avatar_url** (str) – override the default avatar of the bot

async **reply**(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *view=None*, *file=None*, *files=None*, *tts=False*, *hide_user_input=False*, *ephemeral=False*, *delete_after=None*, *allowed_mentions=None*, *type=None*, *fetch_response_message=True*)

Creates an interaction response. What's the difference between this method and *create_response()*? If the token is no longer valid, this method sends a usual channel message instead of creating an interaction response. Also, this method fetches the interaction response message and returns it, unlike *create_response()*.

Parameters

- **content** (str) – message content
- **embed** (discord.Embed) – message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **file** (discord.File) – if it's the first interaction reply, the file will be ignored due to API limitations. Everything else is the same as in discord.TextChannel.send() method.
- **files** (List[discord.File]) – same as file but for multiple files.
- **hide_user_input** (bool) – if set to True, user's input won't be displayed
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **delete_after** (float) – if specified, your reply will be deleted after delete_after seconds
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

- **type** (int | *ResponseType*) – sets the response type. If it's not specified, this method sets it according to `hide_user_input`, `content` and `embed` params.
- **fetch_response_message** (bool) – whether to fetch and return the response message. Defaults to `True`.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both `embed` and `embeds` are specified

Returns **message** – The response message that has been sent or `None` if the message is ephemeral

Return type `discord.Message | None`

1.10.15 ButtonStyle

class dislash.**ButtonStyle**

blurple = 1

grey = 2

green = 3

red = 4

link = 5

1.10.16 CheckAnyFailure

class dislash.**CheckAnyFailure**(*checks*, *errors*)

1.10.17 ClickListener

class dislash.**ClickListener**(*message_id: int*, *timeout: Optional[float] = None*)

Creates a nice click manager for a message. You can use this class, or call `discord.Message.create_click_listener(timeout)` in order to create an instance.

Click manager allows to process button clicks under a specific message using decorator-based interface.

Parameters

- **message_id** (int) – the ID of the message with buttons
- **timeout** (float) – the amount of seconds required after the last matched interaction for the click manager to finish working. Defaults to `None`, which means no timeout.

from_user(*user: discord.user.User*, ***, *cancel_others: bool = False*, *reset_timeout: bool = True*)

A decorator that makes the function below waiting for a click from the specified user.

Parameters are the same as in *matching_condition()*, except `check` parameter is replaced with a `user` to compare with.

kill()

Kills the click manager. Only useful if the `timeout` param was specified as `None`.

matching_condition(*check*, *, *cancel_others: bool = False*, *reset_timeout: bool = True*)
A decorator that makes the function below waiting for a click matching the specified conditions.

Parameters

- **check** (function) – the required condition. This function must take exactly one argument which is guaranteed to be an instance of *MessageInteraction*. This function must return True or False.
- **cancel_others** (bool) – defaults to False. Specifies whether to cancel all other local listeners or not. For example, if this parameter is False, the library will activate all other listeners matching the interaction, untill all listeners are toggled or some of them cancels others.
- **reset_timeout** (bool) – defaults to True. Specifies whether to restart the timer or not. By restarting the timer, you extend the lifespan of all local listeners.

matching_id(*custom_id: str*, *, *cancel_others: bool = False*, *reset_timeout: bool = True*)
A decorator that makes the function below waiting for a click of the button matching the specified custom_id.

Parameters are the same as in *matching_condition()*, except check parameter is replaced with custom_id.

no_checks(*, *cancel_others: bool = False*, *reset_timeout: bool = True*)
A decorator that makes the function below waiting for any click.

Parameters are the same as in *matching_condition()*, except there's no check.

not_from_user(*user: discord.user.User*, *, *cancel_others: bool = False*, *reset_timeout: bool = True*)
A decorator that makes the function below waiting for a click from a user not maching the specified one.

Parameters are the same as in *matching_condition()*, except check parameter is replaced with a user to compare with.

timeout(*func*)
A decorator that makes the function below waiting for click listener timeout.

1.10.18 CommandOnCooldown

class dislash.**CommandOnCooldown**(*cooldown*, *retry_after*)
Exception raised when the application command being invoked is on cooldown.

This inherits from *ApplicationCommandError*

Attributes

cooldown: *Cooldown* (a class with attributes *rate*, *per*, and *type*)

retry_after: *float* (the amount of seconds to wait before you can retry again)

1.10.19 CommandParent

class dislash.**CommandParent**(*func*, *, *name=None*, *description=None*, *options=None*, *default_permission=True*, *guild_ids=None*, *connectors=None*, *auto_sync=True*, ***kwargs*)

error(*func*)
A decorator that makes the function below work as error handler for this command.

get_cooldown_retry_after(*inter*)
Retrieves the amount of seconds before this slash command can be tried again.

Parameters **inter** (*SlashInteraction*) – The interaction to retrieve the cooldown from.

Returns The amount of time left on this slash command's cooldown in seconds. If this is `0.0` then the slash command isn't on cooldown.

Return type `float`

async invoke(*interaction*)

async invoke_children(*interaction*)

is_on_cooldown(*inter*)
Checks whether the slash command is currently on cooldown.

Parameters **inter** (*SlashInteraction*) – The interaction to use when checking the commands cooldown status.

Returns A boolean indicating if the slash command is on cooldown.

Return type `bool`

reset_cooldown(*inter*)
Resets the cooldown on this slash command.

Parameters **inter** (*SlashInteraction*) – The interaction to reset the cooldown under.

sub_command(*name: Optional[str] = None*, *description: Optional[str] = None*, *options: Optional[list] = None*, *connectors: Optional[dict] = None*, ***kwargs*)
A decorator that creates a subcommand under the base command.

Parameters

- **name** (`str`) – the name of the subcommand. Defaults to the function name
- **description** (`str`) – the description of the subcommand
- **options** (`list`) – the options of the subcommand for registration in API
- **connectors** (`dict`) – which function param states for each option. If the name of an option already matches the corresponding function param, you don't have to specify the connectors. Connectors template: `{"option-name": "param_name", ...}`

sub_command_group(*name=None*, ***kwargs*)
A decorator that creates a subcommand group under the base command. Remember that the group must have at least one subcommand.

Parameters **name** (`str`) – the name of the subcommand group. Defaults to the function name

1.10.20 Component

class dislash.**Component**(*type: int*)
The base class for message components

1.10.21 ComponentType

class dislash.**ComponentType**

1.10.22 ContextMenuInteraction

class dislash.**ContextMenuInteraction**(*client*, *payload*)

async **create_response**(*content=None*, ***, *type=None*, *embed=None*, *embeds=None*, *components=None*, *view=None*, *ephemeral=False*, *tts=False*, *allowed_mentions=None*)
Creates an interaction response.

Parameters

- **content** (str) – response content
- **type** (int | *ResponseType*) – sets the response type. See *ResponseType*
- **embed** (discord.Embed) – response embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both embed and embeds are specified

async **delete**()
Deletes the original interaction response.

async **delete_after**(*delay: float*)

async **edit**(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *allowed_mentions=None*)
Edits the original interaction response.

Parameters

- **content** (str) – New message content
- **embed** (discord.Embed) – New message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds of a new message

- **components** (List[ActionRow]) – a list of up to 5 action rows
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Returns **message** – The message that was edited

Return type discord.Message

async fetch_initial_response()
Fetches the original interaction response.

async followup(*content=None*, ***, *embed=None*, *embeds=None*, *file=None*, *files=None*, *components=None*, *view=None*, *tts=False*, *ephemeral=False*, *allowed_mentions=None*, *username=None*, *avatar_url=None*)
Sends a followup message.

Parameters

- **content** (str) – the followup message content
- **embed** (discord.Embed) – the followup message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **file** (discord.File) – a file to attach to the message
- **files** (List[discord.File]) – a list of files to attach to the message
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your message will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message
- **username** (str) – override the default bot name
- **avatar_url** (str) – override the default avatar of the bot

async reply(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *view=None*, *file=None*, *files=None*, *tts=False*, *hide_user_input=False*, *ephemeral=False*, *delete_after=None*, *allowed_mentions=None*, *type=None*, *fetch_response_message=True*)
Creates an interaction response. What's the difference between this method and *create_response()*? If the token is no longer valid, this method sends a usual channel message instead of creating an interaction response. Also, this method fetches the interaction response message and returns it, unlike *create_response()*.

Parameters

- **content** (str) – message content
- **embed** (discord.Embed) – message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation

- **file** (discord.File) – if it's the first interaction reply, the file will be ignored due to API limitations. Everything else is the same as in discord.TextChannel.send() method.
- **files** (List[discord.File]) – same as file but for multiple files.
- **hide_user_input** (bool) – if set to True, user's input won't be displayed
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **delete_after** (float) – if specified, your reply will be deleted after delete_after seconds
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.
- **type** (int | *ResponseType*) – sets the response type. If it's not specified, this method sets it according to hide_user_input, content and embed params.
- **fetch_response_message** (bool) – whether to fetch and return the response message. Defaults to True.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both embed and embeds are specified

Returns **message** – The response message that has been sent or None if the message is ephemeral

Return type discord.Message | None

1.10.23 ContextMenuInteractionData

class dislash.**ContextMenuInteractionData**(*data*, *guild*, *state*)

1.10.24 DiscordException

class dislash.**DiscordException**

Base exception class for discord.py

Ideally speaking, this could be caught to handle any exceptions thrown from this library.

1.10.25 SlashInteraction

class dislash.**SlashInteraction**(*client*, *payload*)

Every interaction with slash-commands is represented by instances of this class

author

The member/user that used the slash-command.

Type discord.Member | discord.User

guild

The guild where interaction was created

Type discord.Guild

channel
The channel where interaction was created

Type discord.TextChannel

data
The arguments that were passed

Type InteractionData

created_at
Then interaction was created

Type datetime.datetime

expired
Whether the interaction token is still valid

Type bool:

async create_response(*content=None*, ***, *type=None*, *embed=None*, *embeds=None*, *components=None*, *view=None*, *ephemeral=False*, *tts=False*, *allowed_mentions=None*)
Creates an interaction response.

Parameters

- **content** (str) – response content
- **type** (int | *ResponseType*) – sets the response type. See *ResponseType*
- **embed** (discord.Embed) – response embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both embed and embeds are specified

async delete()
Deletes the original interaction response.

async delete_after(*delay: float*)

async edit(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *allowed_mentions=None*)
Edits the original interaction response.

Parameters

- **content** (str) – New message content
- **embed** (discord.Embed) – New message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds of a new message

- **components** (List[ActionRow]) – a list of up to 5 action rows
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.

Returns **message** – The message that was edited

Return type discord.Message

async fetch_initial_response()
Fetches the original interaction response.

async followup(*content=None*, ***, *embed=None*, *embeds=None*, *file=None*, *files=None*, *components=None*, *view=None*, *tts=False*, *ephemeral=False*, *allowed_mentions=None*, *username=None*, *avatar_url=None*)
Sends a followup message.

Parameters

- **content** (str) – the followup message content
- **embed** (discord.Embed) – the followup message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **file** (discord.File) – a file to attach to the message
- **files** (List[discord.File]) – a list of files to attach to the message
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **ephemeral** (bool) – if set to True, your message will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message
- **username** (str) – override the default bot name
- **avatar_url** (str) – override the default avatar of the bot

get(*name: str*, *default=None*)
Equivalent to InteractionData.get

get_option(*name: str*)
Equivalent to InteractionData.get_option

option_at(*index: int*)
Equivalent to InteractionData.option_at

async reply(*content=None*, ***, *embed=None*, *embeds=None*, *components=None*, *view=None*, *file=None*, *files=None*, *tts=False*, *hide_user_input=False*, *ephemeral=False*, *delete_after=None*, *allowed_mentions=None*, *type=None*, *fetch_response_message=True*)
Creates an interaction response. What's the difference between this method and *create_response()*? If the token is no longer valid, this method sends a usual channel message instead of creating an interaction response. Also, this method fetches the interaction response message and returns it, unlike *create_response()*.

Parameters

- **content** (str) – message content

- **embed** (discord.Embed) – message embed
- **embeds** (List[discord.Embed]) – a list of up to 10 embeds to attach
- **components** (List[ActionRow]) – a list of up to 5 action rows
- **view** (discord.ui.View) – only usable with discord.py 2.0. Read more about View in discord.py 2.0 official documentation
- **file** (discord.File) – if it's the first interaction reply, the file will be ignored due to API limitations. Everything else is the same as in discord.TextChannel.send() method.
- **files** (List[discord.File]) – same as file but for multiple files.
- **hide_user_input** (bool) – if set to True, user's input won't be displayed
- **ephemeral** (bool) – if set to True, your response will only be visible to the command author
- **tts** (bool) – whether the message is text-to-speech or not
- **delete_after** (float) – if specified, your reply will be deleted after delete_after seconds
- **allowed_mentions** (discord.AllowedMentions) – controls the mentions being processed in this message.
- **type** (int | *ResponseType*) – sets the response type. If it's not specified, this method sets it according to hide_user_input, content and embed params.
- **fetch_response_message** (bool) – whether to fetch and return the response message. Defaults to True.

Raises

- **HTTPException** – sending the response failed
- **InvalidArgument** – Both embed and embeds are specified

Returns **message** – The response message that has been sent or None if the message is ephemeral

Return type discord.Message | None

1.10.26 InteractionCheckFailure

class dislash.**InteractionCheckFailure**(*message=None*, **args*)

1.10.27 InteractionClient

class dislash.**InteractionClient**(*client*, ***, *test_guilds: Optional[List[int]] = None*, *sync_commands: bool = True*, *show_warnings: bool = True*, *modify_send: bool = True*)

The main purpose of this class is to track INTERACTION_CREATE API event.

Parameters

- **client** (commands.Bot | commands.AutoShardedBot) – The discord.py Bot instance
- **show_warnings** (bool) – Whether to show the warnings or not. Defaults to True
- **modify_send** (bool) – Whether to modify Messageable.send and Message.edit. Modified methods allow to specify the components parameter.

client
an instance of any class inherited from `discord.Client`

Type `commands.Bot | commands.AutoShardedBot`

application_id
the ID of the application your bot is related to

Type `int`

global_commands
All registered global application commands

Type List[*ApplicationCommand*]

slash_commands
All invokable slash commands from your code

Type `Dict[str, CommandParent]`

user_commands
All invokable user commands from your code

Type `Dict[str, InvokableUserCommand]`

message_commands
All invokable message commands from your code

Type `Dict[str, InvokableMessageCommand]`

commands
All invokable application commands from your code

Type `Dict[str, InvokableApplicationCommand]`

is_ready
Equals to `True` if SlashClient is ready, otherwise it's `False`

Type bool

async batch_edit_guild_command_permissions(*guild_id: int*, *permissions: dict*)
Batch edits permissions for all commands in a guild.

Parameters

- **guild_id** (`int`) – the ID of the guild
- **permissions** (Dict[`int`, *ApplicationCommandPermissions*]) – a dictionary of command IDs and permissions

async batch_fetch_guild_command_permissions(*guild_id: int*)
Fetches command permissions for all commands in a guild.

Parameters **guild_id** (`int`) – the ID of the guild

async delete_global_command(*command_id: int*)
Deletes the global application command

Parameters **command_id** (*int*) –

async delete_global_command_named(*name: str*)
Deletes a global command matching the specified name.

Parameters **name** (*str*) – the name of the command to delete

async delete_global_commands()
Deletes all global commands.

async delete_guild_command(*guild_id: int*, *command_id: int*)
Deletes the local application command

Parameters

- **guild_id** (*int*) –
- **command_id** (*int*) –

async delete_guild_command_named(*guild_id: int*, *name: str*)
Deletes a local command matching the specified name.

Parameters

- **guild_id** (*int*) – ID of the guild where the command is registered
- **name** (*str*) – the name of the command to edit

async delete_guild_commands(*guild_id: int*)
Deletes all local commands in the specified guild.

Parameters **guild_id** (*int*) – the ID of the guild where you're going to delete the commands

dispatch(*event_name*, **args*, ***kwargs*)

async edit_global_command(*command_id: int*, *app_command:* dislash.interactions.application_command.ApplicationCommand, ***kwargs*)
Edits a global application command

Parameters

- **command_id** (*int*) –
- **app_command** (ApplicationCommand) – replacement of the old data

async edit_global_command_named(*name: str*, *app_command:* dislash.interactions.application_command.ApplicationCommand)
Edits a global command matching the specified name.

Parameters

- **name** (*str*) – the name of the command to edit
- **app_command** (ApplicationCommand) – replacement of the old data

async edit_guild_command(*guild_id: int*, *command_id: int*, *app_command:* dislash.interactions.application_command.ApplicationCommand, ***kwargs*)
Edits the local application command

Parameters

- **guild_id** (*int*) –
- **command_id** (*int*) –
- **app_command** (ApplicationCommand) – replacement of the old data

async edit_guild_command_named(*guild_id: int*, *name: str*, *app_command:* dislash.interactions.application_command.ApplicationCommand)
Edits a local command matching the specified name.

Parameters

- **guild_id** (*int*) – ID of the guild where the command is registered
- **name** (*str*) – the name of the command to edit
- **app_command** (ApplicationCommand) – replacement of the old data

async edit_guild_command_permissions(*guild_id: int*, *command_id: int*, *permissions:* dislash.interactions.application_command.ApplicationCommandPermissions)

Edits command permissions for a specific command in a guild.

Parameters

- **guild_id** (int) – the ID of the guild
- **command_id** (int) – the ID of the command
- **permissions** (SlashCommandPermissions | dict) – new permissions to set. If you use SlashCommandPermissions.from_pairs, you can pass the arg of that method straight into this function

event(*func*)

Decorator

```
@slash.event
async def on_ready():
    print("SlashClient is ready")
```

All possible events:

on_ready, on_auto_register,

on_slash_command, on_slash_command_error

async fetch_global_command(*command_id: int*)

Requests a registered global command

Parameters **command_id** (*int*) –

Returns **global_command**

Return type *ApplicationCommand*

async fetch_global_command_named(*name: str*)

Fetches a global command that matches the specified name

Parameters **name** (*str*) – the name of the command to fetch

async fetch_global_commands()

Requests a list of global registered commands from the API

Returns **global_commands**

Return type List[*ApplicationCommand*]

async fetch_guild_command(*guild_id: int*, *command_id: int*)

Requests a registered guild command

Parameters

- **guild_id** (*int*) –
- **command_id** (*int*) –

Returns **guild_command**

Return type *ApplicationCommand*

async fetch_guild_command_named(*guild_id: int*, *name: str*)

Fetches a guild command that matches the specified name

Parameters

- **guild_id** (*int*) – ID of the guild where the command is registered
- **name** (*str*) – the name of the command to fetch

async fetch_guild_command_permissions(*guild_id: int*, *command_id: int*)
Fetches command permissions for a specific command in a guild.

Parameters

- **guild_id** (int) – the ID of the guild
- **command_id** (int) – the ID of the command

async fetch_guild_commands(*guild_id: int*)
Requests a list of registered commands for a specific guild

Parameters **guild_id** (*int*) –

Returns guild_commands

Return type List[*ApplicationCommand*]

get_global_command(*command_id: int*)
Get a cached global command

Parameters **command_id** (*int*) – the ID of the command

Returns slash_command

Return type *SlashCommand* | None

get_global_command_named(*name: str*)
Get a cached global command matching the specified name

Parameters **name** (*str*) – the name of the command

Returns slash_command

Return type *SlashCommand* | None

get_guild_command(*guild_id: int*, *command_id: int*)
Get a cached guild command

Parameters

- **guild_id** (*int*) – the ID of the guild
- **command_id** (*int*) – the ID of the command

Returns slash_command

Return type *SlashCommand* | None

get_guild_command_named(*guild_id: int*, *name: str*)
Get a cached guild command matching the specified name

Parameters

- **guild_id** (*int*) – the ID of the guild
- **name** (*str*) – the name of the command

Returns slash_command

Return type *SlashCommand* | None

get_guild_commands(*guild_id: int*)
Get cached guild commands

Parameters **guild_id** (*int*) – the ID of the guild

Returns

Return type ~:class:*List[ApplicationCommand]*

message_command(**args*, ***kwargs*)

async multiple_wait_for(*events_and_checks: Dict[str, Any]*, *timeout: Optional[float] = None*)
Waits until one of the given events toggles and matches the relevant check.

Example:

```
result = None
try:
    result = await client.multiple_wait_for(
        {
            "message": lambda msg: msg.author == ctx.author,
            "reaction_add": lambda react, user: user == ctx.author
        },
        timeout=60
    )
except asyncio.TimeoutError:
    await ctx.send("It took too long")
if isinstance(result, discord.Message):
    # on_message was toggled
    await ctx.send(f"You said '{result.content}'")
else:
    # on_reaction_add was toggled
    reaction, user = result
    await ctx.send(f"Your reaction: {reaction.emoji}")
```

Parameters

- **events_and_checks** (Dict[str, function | None]) – a dictionary of event names and relevant checks, e.g. {"message": lambda m: m.author == ctx.author, "button_click": None}
- **timeout** (float | None) – the amount of seconds the bot should wait for any of the given events

async overwrite_global_commands(*app_commands: list*)
Bulk overwrites all global application commands

Parameters **app_commands** (*List[ApplicationCommand]*) –

async overwrite_guild_commands(*guild_id: int*, *app_commands: list*)
Bulk overwrites all guild application commands

Parameters

- **guild_id** (*int*) –
- **app_commands** (*List[ApplicationCommand]*) –

async register_global_command(*app_command:* dislash.interactions.application_command.ApplicationCommand)
Registers a global application command

Parameters **app_command** (ApplicationCommand) –

async register_guild_command(*guild_id: int*, *app_command: dislash.interactions.application_command.ApplicationCommand*)

Registers a local application command

Parameters

- **guild_id** (int) –
- **app_command** (*ApplicationCommand*) –

slash_command(**args*, ***kwargs*)

A decorator that allows to build a slash command.

Parameters

- **auto_sync** (bool) – whether to automatically register the command or not. Defaults to True
- **name** (str) – name of the slash command you want to respond to (equals to function name by default).
- **description** (str) – the description of the slash command. It will be visible in Discord.
- **options** (List[Option]) – the list of slash command options. The options will be visible in Discord.
- **default_permission** (bool) – whether the command is enabled by default when the app is added to a guild.
- **guild_ids** (List[int]) – if specified, the client will register a command in these guilds. Otherwise this command will be registered globally.
- **connectors** (dict) – which function param states for each option. If the name of an option already matches the corresponding function param, you don't have to specify the connectors. Connectors template: {"option-name": "param_name", ...}

teardown()

Cleanup the client by removing all registered listeners and caches.

user_command(**args*, ***kwargs*)

async wait_for_button_click(*check=None*, *timeout=None*)

async wait_for_dropdown(*check=None*, *timeout=None*)

1.10.28 InteractionDataOption

class dislash.**InteractionDataOption**(***, *data*, *resolved: dislash.interactions.app_command_interaction.Resolved*)

Represents user's input for a specific option

name

The name of the option

Type str

value

The value of the option

Type Any

`options`

Represents options of a sub-slash-command.
{`name`: *InteractionDataOption*, ...}

Type dict

`get`(*name: str*, *default=None*)
Get the value of an option with the specified name

Parameters

- **name** (*str*) – the name of the option you want to get
- **default** (*any*) – what to return in case nothing was found

Returns

- **option_value** (*any*) – The option type isn't `SUB_COMMAND_GROUP` or `SUB_COMMAND`
- **option** (InteractionDataOption | `default`) – Otherwise

`get_option`(*name: str*)
Get the raw *InteractionDataOption* matching the specified name

Parameters **name** (*str*) – The name of the option you want to get

Returns **option**

Return type InteractionDataOption | `None`

`option_at`(*index: int*)
Similar to `InteractionData.option_at`

1.10.29 InteractionType

`class dislash.InteractionType`

1.10.30 InvokableApplicationCommand

`class dislash.InvokableApplicationCommand`(*func*, *, *name=None*, ***kwargs*)

`error`(*func*)
A decorator that makes the function below work as error handler for this command.

`get_cooldown_retry_after`(*inter*)
Retrieves the amount of seconds before this slash command can be tried again.

Parameters **inter** (*SlashInteraction*) – The interaction to retrieve the cooldown from.

Returns The amount of time left on this slash command's cooldown in seconds. If this is `0.0` then the slash command isn't on cooldown.

Return type `float`

`is_on_cooldown`(*inter*)
Checks whether the slash command is currently on cooldown.

Parameters **inter** (*SlashInteraction*) – The interaction to use when checking the commands cooldown status.

Returns A boolean indicating if the slash command is on cooldown.

Return type bool

reset_cooldown(*inter*)
Resets the cooldown on this slash command.

Parameters **inter** (*SlashInteraction*) – The interaction to reset the cooldown under.

1.10.31 InvokableContextMenuCommand

class dislash.**InvokableContextMenuCommand**(*func*, *, *name=None*, *guild_ids=None*, ***kwargs*)

error(*func*)
A decorator that makes the function below work as error handler for this command.

get_cooldown_retry_after(*inter*)
Retrieves the amount of seconds before this slash command can be tried again.

Parameters **inter** (*SlashInteraction*) – The interaction to retrieve the cooldown from.

Returns The amount of time left on this slash command's cooldown in seconds. If this is `0.0` then the slash command isn't on cooldown.

Return type float

async invoke(*interaction*)

is_on_cooldown(*inter*)
Checks whether the slash command is currently on cooldown.

Parameters **inter** (*SlashInteraction*) – The interaction to use when checking the commands cooldown status.

Returns A boolean indicating if the slash command is on cooldown.

Return type bool

reset_cooldown(*inter*)
Resets the cooldown on this slash command.

Parameters **inter** (*SlashInteraction*) – The interaction to reset the cooldown under.

1.10.32 InvokableMessageCommand

class dislash.**InvokableMessageCommand**(*func*, *, *name=None*, *guild_ids=None*, ***kwargs*)

error(*func*)
A decorator that makes the function below work as error handler for this command.

get_cooldown_retry_after(*inter*)
Retrieves the amount of seconds before this slash command can be tried again.

Parameters **inter** (*SlashInteraction*) – The interaction to retrieve the cooldown from.

Returns The amount of time left on this slash command's cooldown in seconds. If this is `0.0` then the slash command isn't on cooldown.

Return type float

async invoke(*interaction*)

is_on_cooldown(*inter*)
Checks whether the slash command is currently on cooldown.

Parameters **inter** (*SlashInteraction*) – The interaction to use when checking the commands cooldown status.

Returns A boolean indicating if the slash command is on cooldown.

Return type bool

reset_cooldown(*inter*)
Resets the cooldown on this slash command.

Parameters **inter** (*SlashInteraction*) – The interaction to reset the cooldown under.

1.10.33 InvokableUserCommand

class dislash.**InvokableUserCommand**(*func*, *, *name=None*, *guild_ids=None*, ***kwargs*)

error(*func*)
A decorator that makes the function below work as error handler for this command.

get_cooldown_retry_after(*inter*)
Retrieves the amount of seconds before this slash command can be tried again.

Parameters **inter** (*SlashInteraction*) – The interaction to retrieve the cooldown from.

Returns The amount of time left on this slash command's cooldown in seconds. If this is 0.0 then the slash command isn't on cooldown.

Return type float

async invoke(*interaction*)

is_on_cooldown(*inter*)
Checks whether the slash command is currently on cooldown.

Parameters **inter** (*SlashInteraction*) – The interaction to use when checking the commands cooldown status.

Returns A boolean indicating if the slash command is on cooldown.

Return type bool

reset_cooldown(*inter*)
Resets the cooldown on this slash command.

Parameters **inter** (*SlashInteraction*) – The interaction to reset the cooldown under.

1.10.34 SelectOption

class dislash.**SelectOption**(*label: str*, *value: str*, *description: Optional[str] = None*, *emoji: Optional[str] = None*, *default: bool = False*)

This class represents an option in a select menu.

Parameters

- **label** (str) – the user-facing name of the option, max 25 characters
- **value** (str) – the dev-define value of the option, max 100 characters
- **description** (str) – an additional description of the option, max 50 characters
- **emoji** (str) – well add an emoji to the option
- **default** (bool) – will render this option as selected by default

classmethod from_dict(*data: dict*)

to_dict()

1.10.35 MessageCommand

class dislash.**MessageCommand**(*name: str*, ***kwargs*)

classmethod from_dict(*data: dict*)

to_dict(***kwargs*)

1.10.36 MissingAnyRole

class dislash.**MissingAnyRole**(*missing_roles*)

1.10.37 MissingPermissions

class dislash.**MissingPermissions**(*missing_perms*, **args*)

1.10.38 MissingRole

class dislash.**MissingRole**(*missing_role*)

1.10.39 NSFWChannelRequired

class dislash.**NSFWChannelRequired**(*channel*)

1.10.40 NoPrivateMessage

class dislash.**NoPrivateMessage**(*message=None*)

1.10.41 NotGuildOwner

class dislash.**NotGuildOwner**(*message=None*, **args*)

1.10.42 NotOwner

class dislash.**NotOwner**(*message=None*, **args*)

1.10.43 Option

class dislash.**Option**(*name: str*, *description: Optional[str] = None*, *type: Optional[int] = None*, *required: bool = False*, *choices: Optional[List[*dislash.interactions.application_command.OptionChoice*]] = None*, *options: Optional[list] = None*)

Parameters

- **name** (str) – option's name
- **description** (str) – option's description
- **type** (Type) – the option type, e.g. `Type.USER`, see *OptionType*
- **required** (bool) – whether this option is required or not
- **choices** (List[*OptionChoice*]) – the list of option choices, type *OptionChoice*
- **options** (List[*Option*]) – the list of sub options. You can only specify this parameter if the `type` is `Type.SUB_COMMAND` or `Type.SUB_COMMAND_GROUP`

add_choice(*name: str*, *value: Any*)

Adds an OptionChoice to the list of current choices

Parameters are the same as for *OptionChoice*

add_option(*name: str*, *description: Optional[str] = None*, *type: Optional[int] = None*, *required: bool = False*, *choices: Optional[List[*dislash.interactions.application_command.OptionChoice*]] = None*, *options: Optional[list] = None*)

Adds an option to the current list of options

Parameters are the same as for *Option*

classmethod from_dict(*payload: dict*)

to_dict()

1.10.44 OptionChoice

class dislash.**OptionChoice**(*name: str*, *value: Any*)

Parameters

- **name** (*str*) – the name of the option-choice (visible to users)
- **value** (*str or int*) – the value of the option-choice

1.10.45 OptionType

class dislash.**OptionType**

SUB_COMMAND = 1

SUB_COMMAND_GROUP = 2

STRING = 3

INTEGER = 4

BOOLEAN = 5

USER = 6

CHANNEL = 7

ROLE = 8

MENTIONABLE = 9

NUMBER = 10

1.10.46 PrivateMessageOnly

class dislash.**PrivateMessageOnly**(*message=None*)

1.10.47 RawCommandPermission

class dislash.**RawCommandPermission**(*id: int*, *type: int*, *permission: bool*)

Represents command permissions for a role or a user.

id

ID of a target

Type int

type

1 if target is a role; 2 if target is a user

Type int

permission

allow or deny the access to the command

Type bool

classmethod from_dict(*data: dict*)

classmethod from_pair(*target: Union[discord.role.Role, discord.user.User]*, *permission: bool*)

to_dict()

1.10.48 ResponseType

class dislash.**ResponseType**

All possible response type values. Used in `Interaction.reply`

Pong = 1

ACK a Ping

ChannelMessageWithSource = 4

Respond to an interaction with a message

AcknowledgeWithSource = 5

ACK an interaction and edit a response later, the user sees a loading state

DeferredUpdateMessage = 6

For components, ACK an interaction and edit the original message later; the user does not see a loading state

UpdateMessage = 7

For components, edit the message the component was attached to

1.10.49 SelectMenu

class dislash.**SelectMenu**(**, custom_id: Optional[str] = None, placeholder: Optional[str] = None, min_values: int = 1, max_values: int = 1, options: Optional[list] = None, disabled: bool = False*)

This class represents a select menu.

Parameters

- **custom_id** (`str`) – a developer-defined identifier for the button, max 100 characters
- **placeholder** (`str`) – custom placeholder text if nothing is selected, max 100 characters
- **min_values** (`int`) – the minimum number of items that must be chosen; default 1, min 0, max 25
- **max_values** (`int`) – the maximum number of items that can be chosen; default 1, max 25
- **options** (List[*SelectOption*]) – the choices in the select, max 25
- **disabled** (`bool`) – disable the menu, defaults to false

custom_id

a developer-defined identifier for the button, max 100 characters

Type `str`

placeholder

custom placeholder text if nothing is selected, max 100 characters

Type `str`

min_values

the minimum number of items that must be chosen; default 1, min 0, max 25

Type `int`

max_values
the maximum number of items that can be chosen; default 1, max 25

Type int

options
the choices in the select, max 25

Type List[*SelectOption*]

disabled
disable the menu, defaults to false

Type bool

selected_options
the list of chosen options, max 25

Type List[*SelectOption*]

add_option(*label: str*, *value: str*, *description: Optional[str] = None*, *emoji: Optional[str] = None*, *default: bool = False*)
Adds an option to the list of options of the menu. Parameters are the same as in *SelectOption*.

classmethod from_dict(*data: dict*)

to_dict()

1.10.50 SlashCommand

class dislash.**SlashCommand**(*name: str*, *description: str*, *options: Optional[list] = None*, *default_permission: bool = True*, ***kwargs*)
A base class for building slash-commands.

Parameters

- **name** (str) – The command name
- **description** (str) – The command description (it'll be displayed by discord)
- **options** (*List[Option]*) – The options of the command. See *Option*
- **default_permission** (bool) – Whether the command is enabled by default when the app is added to a guild

add_option(*name: str*, *description: Optional[str] = None*, *type: Optional[int] = None*, *required: bool = False*, *choices: Optional[List[*dislash.interactions.application_command.OptionChoice*]] = None*, *options: Optional[list] = None*)
Adds an option to the current list of options

Parameters are the same as for *Option*

classmethod from_dict(*payload: dict*)

to_dict(*, *hide_name=False*)

1.10.51 SlashInteractionData

class dislash.**SlashInteractionData**(*, *data*, *guild*, *state*)

id
The id of the interaction

Type int

name
The name of activated slash-command

Type str

options

Represents options of the slash-command.
{name: *InteractionDataOption*, ...}

Type dict

resolved
The collection of related objects, such as users, members, roles, channels and messages

Type Resolved

get(*name: str*, *default=None*)
Get the value of an option with the specified name

Parameters

- **name** (*str*) – the name of the option you want to get
- **default** (*any*) – what to return in case nothing was found

Returns

- **option_value** (*any*) – The option type isn't SUB_COMMAND_GROUP or SUB_COMMAND
- **option** (*InteractionDataOption* | default) – Otherwise

get_option(*name: str*)
Get the raw *InteractionDataOption* matching the specified name

Parameters **name** (*str*) – The name of the option you want to get

Returns **option**

Return type *InteractionDataOption* | None

option_at(*index: int*)
Get an option by it's index

Parameters **index** (*int*) – the index of the option you want to get

Returns **option** – the option located at the specified index

Return type *InteractionDataOption* | None

1.10.52 SubCommand

class dislash.**SubCommand**(*func*, *, *name=None*, *description=None*, *options=None*, *connectors=None*, ***kwargs*)

error(*func*)
A decorator that makes the function below work as error handler for this command.

get_cooldown_retry_after(*inter*)
Retrieves the amount of seconds before this slash command can be tried again.

Parameters **inter** (*SlashInteraction*) – The interaction to retrieve the cooldown from.

Returns The amount of time left on this slash command's cooldown in seconds. If this is `0.0` then the slash command isn't on cooldown.

Return type `float`

is_on_cooldown(*inter*)
Checks whether the slash command is currently on cooldown.

Parameters **inter** (*SlashInteraction*) – The interaction to use when checking the commands cooldown status.

Returns A boolean indicating if the slash command is on cooldown.

Return type `bool`

reset_cooldown(*inter*)
Resets the cooldown on this slash command.

Parameters **inter** (*SlashInteraction*) – The interaction to reset the cooldown under.

1.10.53 SubCommandGroup

class dislash.**SubCommandGroup**(*func*, *, *name=None*, ***kwargs*)

error(*func*)
A decorator that makes the function below work as error handler for this command.

get_cooldown_retry_after(*inter*)
Retrieves the amount of seconds before this slash command can be tried again.

Parameters **inter** (*SlashInteraction*) – The interaction to retrieve the cooldown from.

Returns The amount of time left on this slash command's cooldown in seconds. If this is `0.0` then the slash command isn't on cooldown.

Return type `float`

is_on_cooldown(*inter*)
Checks whether the slash command is currently on cooldown.

Parameters **inter** (*SlashInteraction*) – The interaction to use when checking the commands cooldown status.

Returns A boolean indicating if the slash command is on cooldown.

Return type `bool`

reset_cooldown(*inter*)
Resets the cooldown on this slash command.

Parameters **inter** (*SlashInteraction*) – The interaction to reset the cooldown under.

sub_command(*name: Optional[str] = None*, *description: Optional[str] = None*, *options: Optional[list] = None*, *connectors: Optional[dict] = None*, ***kwargs*)
A decorator that creates a subcommand in the subcommand group.

Parameters are the same as in *CommandParent.sub_command*

1.10.54 UserCommand

class dislash.**UserCommand**(*name: str*, ***kwargs*)

classmethod from_dict(*data: dict*)

to_dict(***kwargs*)

CHAPTER

TWO

LINKS

- search
- PyPi
- GitHub
- Discord

INDEX

E

F

G

H

I

K

M

N

O

P

R

S

T

U

V

W