aiogram Documentation

Release 3.13.1

aiogram Team

Sep 27, 2024

CONTENTS

aiogram is a modern and fully asynchronous framework for Telegram Bot API written in Python 3.8+ using asyncio and aiohttp.

Make your bots faster and more powerful!

Documentation:

- English
- Ukrainian

CHAPTER

ONE

FEATURES

- Asynchronous (asyncio docs, **PEP 492**)
- Has type hints (**PEP 484**) and can be used with mypy
- Supports PyPy
- Supports Telegram Bot API 7.10 and gets fast updates to the latest versions of the Bot API
- Telegram Bot API integration code was autogenerated and can be easily re-generated when API gets updated
- Updates router (Blueprints)
- Has Finite State Machine
- Uses powerful magic filters
- Middlewares (incoming updates and API calls)
- Provides Replies into Webhook
- Integrated I18n/L10n support with GNU Gettext (or Fluent)

Warning

It is strongly advised that you have prior experience working with asyncio before beginning to use **aiogram**.

If you have any questions, you can visit our community chats on Telegram:

- @aiogram
- @aiogramua
- @aiogram_uz
- @aiogram_kz
- @aiogram_ru
- @aiogram_fa
- @aiogram_it
- @aiogram_br

1.1 Simple usage

```
import asyncio
import logging
import sys
from os import getenv

from aiogram import Bot, Dispatcher, html
from aiogram.client.default import DefaultBotProperties
from aiogram.enums import ParseMode
from aiogram.filters import CommandStart
from aiogram.types import Message

# Bot token can be obtained via https://t.me/BotFather
TOKEN = getenv("BOT_TOKEN")

# All handlers should be attached to the Router (or Dispatcher)

dp = Dispatcher()


@dp.message(CommandStart())
async def command_start_handler(message: Message) -> None:
    """
    This handler receives messages with `/start` command
    """
    # Most event objects have aliases for API methods that can be called in events'␣
↪context
    # For example if you want to answer to incoming message you can use `message.answer(.
↪..)` alias
    # and the target chat will be passed to :ref:`aiogram.methods.send_message.
↪SendMessage`
    # method automatically or call API method directly via
    # Bot instance: `bot.send_message(chat_id=message.chat.id, ...)`
    await message.answer(f"Hello, {html.bold(message.from_user.full_name)}!")


@dp.message()
async def echo_handler(message: Message) -> None:
    """
    Handler will forward receive a message back to the sender

    By default, message handler will handle all message types (like a text, photo,␣
↪sticker etc.)
    """
    try:
        # Send a copy of the received message
        await message.send_copy(chat_id=message.chat.id)
    except TypeError:
        # But not all the types is supported to be copied so need to handle it
        await message.answer("Nice try!")

```

(continues on next page)

(continued from previous page)

```

async def main() -> None:
    # Initialize Bot instance with default bot properties which will be passed to all␣
↪API calls
    bot = Bot(token=TOKEN, default=DefaultBotProperties(parse_mode=ParseMode.HTML))

    # And the run events dispatching
    await dp.start_polling(bot)


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, stream=sys.stdout)
    asyncio.run(main())
```

1.2 Usage without dispatcher

Just only interact with Bot API, without handling events

```
import asyncio
from argparse import ArgumentParser

from aiogram import Bot
from aiogram.client.default import DefaultBotProperties
from aiogram.enums import ParseMode


def create_parser() -> ArgumentParser:
    parser = ArgumentParser()
    parser.add_argument("--token", help="Telegram Bot API Token")
    parser.add_argument("--chat-id", type=int, help="Target chat id")
    parser.add_argument("--message", "-m", help="Message text to sent", default="Hello,␣
↪World!")

    return parser


async def main():
    parser = create_parser()
    ns = parser.parse_args()

    token = ns.token
    chat_id = ns.chat_id
    message = ns.message

    async with Bot(
        token=token,
        default=DefaultBotProperties(
            parse_mode=ParseMode.HTML,
        ),
    ) as bot:
```

(continues on next page)

(continued from previous page)

```
        await bot.send_message(chat_id=chat_id, text=message)


if __name__ == "__main__":
    asyncio.run(main())
```

CHAPTER

TWO

CONTENTS

2.1 Installation

2.1.1 From PyPI

```
pip install -U aiogram
```

2.1.2 From Arch Linux Repository

Warning

Package in this repository may be outdated. Use PyPI package for the latest version.

```
pacman -S python-aiogram
```

Development build

2.1.3 From GitHub

```
pip install https://github.com/aiogram/aiogram/archive/refs/heads/dev-3.x.zip
```

2.2 Migration FAQ (2.x -> 3.0)

Danger

This guide is still in progress.

This version introduces numerous breaking changes and architectural improvements. It helps reduce the count of global variables in your code, provides useful mechanisms to modularize your code, and enables the creation of shareable modules via packages on PyPI. It also makes middlewares and filters more controllable, among other improvements.

On this page, you can read about the changes made in relation to the last stable 2.x version.

Note

This page more closely resembles a detailed changelog than a migration guide, but it will be updated in the future.

Feel free to contribute to this page, if you find something that is not mentioned here.

2.2.1 Dependencies

- The dependencies required for `i18n` are no longer part of the default package. If your application uses translation functionality, be sure to add an optional dependency:

  ```
  pip install aiogram[i18n]
  ```

2.2.2 Dispatcher

- The *`Dispatcher`* class no longer accepts a `Bot` instance in its initializer. Instead, the `Bot` instance should be passed to the dispatcher only for starting polling or handling events from webhooks. This approach also allows for the use of multiple bot instances simultaneously ("multibot").
- *`Dispatcher`* now can be extended with another Dispatcher-like thing named *`Router`*. With routes, you can easily modularize your code and potentially share these modules between projects. (*Read more »*.)
- Removed the **_handler** suffix from all event handler decorators and registering methods. (*Read more »*)
- The `Executor` has been entirely removed; you can now use the *`Dispatcher`* directly to start poll the API or handle webhooks from it.
- The throttling method has been completely removed; you can now use middlewares to control the execution context and implement any throttling mechanism you desire.
- Removed global context variables from the API types, `Bot` and *`Dispatcher`* object. From now on, if you want to access the current bot instance within handlers or filters, you should accept the argument `bot: Bot` and use it instead of `Bot.get_current()`. In middlewares, it can be accessed via `data["bot"]`.
- To skip pending updates, you should now call the *`DeleteWebhook`* method directly, rather than passing `skip_updates=True` to the start polling method.
- To feed updates to the *`Dispatcher`*, instead of method `process_update()`, you should use method *`feed_update()`*. (*Read more »*)

2.2.3 Filtering events

- Keyword filters can no longer be used; use filters explicitly. (Read more »)
- Due to the removal of keyword filters, all previously enabled-by-default filters (such as state and content_type) are now disabled. You must specify them explicitly if you wish to use them. For example instead of using `@dp.message_handler(content_types=ContentType.PHOTO)` you should use `@router.message(F.photo)`
- Most common filters have been replaced with the "magic filter." (*Read more »*)
- By default, the message handler now receives any content type. If you want a specific one, simply add the appropriate filters (Magic or any other).
- The state filter is no longer enabled by default. This means that if you used `state="*"` in v2, you should not pass any state filter in v3. Conversely, if the state was not specified in v2, you will now need to specify it in v3.

- Added the possibility to register global filters for each router, which helps to reduce code repetition and provides an easier way to control the purpose of each router.

2.2.4 Bot API

- All API methods are now classes with validation, implemented via *pydantic <https://docs.pydantic.dev/>*. These API calls are also available as methods in the Bot class.
- More pre-defined Enums have been added and moved to the *aiogram.enums* sub-package. For example, the chat type enum is now `aiogram.enums.ChatType` instead of `aiogram.types.chat.ChatType`.
- The HTTP client session has been separated into a container that can be reused across different Bot instances within the application.
- API Exceptions are no longer classified by specific messages, as Telegram has no documented error codes. However, all errors are classified by HTTP status codes, and for each method, only one type of error can be associated with a given code. Therefore, in most cases, you should check only the error type (by status code) without inspecting the error message.

2.2.5 Middlewares

- Middlewares can now control an execution context, e.g., using context managers. (*Read more »*)
- All contextual data is now shared end-to-end between middlewares, filters, and handlers. For example now you can easily pass some data into context inside middleware and get it in the filters layer as the same way as in the handlers via keyword arguments.
- Added a mechanism named **flags** that helps customize handler behavior in conjunction with middlewares. (*Read more »*)

2.2.6 Keyboard Markup

- Now `aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup` and `aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup` no longer have methods for extension, instead you have to use markup builders `aiogram.utils.keyboard.ReplyKeyboardBuilder` and `aiogram.utils.keyboard.KeyboardBuilder` respectively (*Read more »*)

2.2.7 Callbacks data

- The callback data factory is now strictly typed using pydantic models. (*Read more »*)

2.2.8 Finite State machine

- State filters will no longer be automatically added to all handlers; you will need to specify the state if you want to use it.
- Added the possibility to change the FSM strategy. For example, if you want to control the state for each user based on chat topics rather than the user in a chat, you can specify this in the `Dispatcher`.
- Now `aiogram.fsm.state.State` and `aiogram.fsm.state.StateGroup` don't have helper methods like `.set()`, `.next()`, etc. Instead, you should set states by passing them directly to `aiogram.fsm.context.FSMContext` (*Read more »*)

- The state proxy is deprecated; you should update the state data by calling `state.set_data(...)` and `state.get_data()` respectively.

2.2.9 Sending Files

- From now on, you should wrap files in an InputFile object before sending them, instead of passing the IO object directly to the API method. (*Read more »*)

2.2.10 Webhook

- The aiohttp web app configuration has been simplified.
- By default, the ability to upload files has been added when you make requests in response to updates (available for webhook only).

2.2.11 Telegram API Server

- The `server` parameter has been moved from the `Bot` instance to `api` parameter of the *`BaseSession`*.
- The constant `aiogram.bot.api.TELEGRAM_PRODUCTION` has been moved to `aiogram.client.telegram.PRODUCTION`.

2.2.12 Telegram objects transformation (to dict, to json, from json)

- Methods `TelegramObject.to_object()`, `TelegramObject.to_json()` and `TelegramObject.to_python()` have been removed due to the use of pydantic models.
- `TelegramObject.to_object()` should be replaced by `TelegramObject.model_validate()` (Read more)
- `TelegramObject.as_json()` should be replaced by *`aiogram.utils.serialization.deserialize_telegram_object_to_python()`*
- `<TelegramObject>.to_python()` should be replaced by `json.dumps(deserialize_telegram_object_to_python(<Tele`

Here are some usage examples:

- Creating an object from a dictionary representation of an object

```
# Version 2.x
message_dict = {"id": 42, ...}
message_obj = Message.to_object(message_dict)
print(message_obj)
# id=42 name='n' ...
print(type(message_obj))
# <class 'aiogram.types.message.Message'>
```

```
# Version 3.x
message_dict = {"id": 42, ...}
message_obj = Message.model_validate(message_dict)
print(message_obj)
# id=42 name='n' ...
print(type(message_obj))
# <class 'aiogram.types.message.Message'>
```

- Creating a json representation of an object

```python
# Version 2.x
async def handler(message: Message) -> None:
    message_json = message.as_json()
    print(message_json)
    # {"id": 42, ...}
    print(type(message_json))
    # <class 'str'>
```

```python
# Version 3.x
async def handler(message: Message) -> None:
    message_json = json.dumps(deserialize_telegram_object_to_python(message))
    print(message_json)
    # {"id": 42, ...}
    print(type(message_json))
    # <class 'str'>
```

- Creating a dictionary representation of an object

```python
async def handler(message: Message) -> None:
    # Version 2.x
    message_dict = message.to_python()
    print(message_dict)
    # {"id": 42, ...}
    print(type(message_dict))
    # <class 'dict'>
```

```python
async def handler(message: Message) -> None:
    # Version 3.x
    message_dict = deserialize_telegram_object_to_python(message)
    print(message_dict)
    # {"id": 42, ...}
    print(type(message_dict))
    # <class 'dict'>
```

2.2.13 ChatMember tools

- Now *aiogram.types.chat_member.ChatMember* no longer contains tools to resolve an object with the appropriate status.

```python
# Version 2.x
from aiogram.types import ChatMember

chat_member = ChatMember.resolve(**dict_data)
```

```python
# Version 3.x
from aiogram.utils.chat_member import ChatMemberAdapter

chat_member = ChatMemberAdapter.validate_python(dict_data)
```

- Now *aiogram.types.chat_member.ChatMember* and all its child classes no longer contain methods for checking for membership in certain logical groups. As a substitute, you can use pre-defined groups or create such groups yourself and check their entry using the `isinstance()` function

```python
# Version 2.x

if chat_member.is_chat_admin():
    print("ChatMember is chat admin")

if chat_member.is_chat_member():
    print("ChatMember is in the chat")
```

```python
# Version 3.x

from aiogram.utils.chat_member import ADMINS, MEMBERS

if isinstance(chat_member, ADMINS):
    print("ChatMember is chat admin")

if isinstance(chat_member, MEMBERS):
    print("ChatMember is in the chat")
```

Note

You also can independently create group similar to ADMINS that fits the logic of your application.

E.g., you can create a PUNISHED group and include banned and restricted members there!

2.3 Bot API

aiogram now is fully support of Telegram Bot API

All methods and types is fully autogenerated from Telegram Bot API docs by parser with code-generator.

2.3.1 Bot

Bot instance can be created from `aiogram.Bot` (`from aiogram import Bot`) and you can't use methods without instance of bot with configured token.

This class has aliases for all methods and named in `lower_camel_case`.

For example `sendMessage` named `send_message` and has the same specification with all class-based methods.

Warning

A full list of methods can be found in the appropriate section of the documentation

class `aiogram.client.bot.`**`Bot`**(*token: str*, *session:* BaseSession | *None = None*, *default:* DefaultBotProperties | *None = None*, ***kwargs: Any*)

Bases: `object`

`__init__`(*token: str*, *session:* BaseSession | *None = None*, *default:* DefaultBotProperties | *None = None*, ***kwargs: Any*) → None

Bot class

Parameters

- **`token`** – Telegram Bot token Obtained from @BotFather
- **`session`** – HTTP Client session (For example AiohttpSession). If not specified it will be automatically created.
- **`default`** – Default bot properties. If specified it will be propagated into the API methods at runtime.

Raises

`TokenValidationError` – When token has invalid format this exception will be raised

`property token: str`

`property id: int`

Get bot ID from token

Returns

`context`(*auto_close: bool = True*) → AsyncIterator[Bot]

Generate bot context

Parameters

`auto_close` – close session on exit

Returns

`async me`() → *User*

Cached alias for getMe method

Returns

`async download_file`(*file_path: str*, *destination: BinaryIO | Path | str | None = None*, *timeout: int = 30*, *chunk_size: int = 65536*, *seek: bool = True*) → BinaryIO | None

Download file by file_path to destination.

If you want to automatically create destination (`io.BytesIO`) use default value of destination and handle result of this method.

Parameters

- **`file_path`** – File path on Telegram server (You can get it from `aiogram.types.File`)
- **`destination`** – Filename, file path or instance of `io.IOBase`. For e.g. `io.BytesIO`, defaults to None
- **`timeout`** – Total timeout in seconds, defaults to 30
- **`chunk_size`** – File chunks size, defaults to 64 kb
- **`seek`** – Go to start of file when downloading is finished. Used only for destination with `typing.BinaryIO` type, defaults to True

`async download`(*file: str | Downloadable*, *destination: BinaryIO | Path | str | None = None*, *timeout: int = 30*, *chunk_size: int = 65536*, *seek: bool = True*) → BinaryIO | None

Download file by file_id or Downloadable object to destination.

If you want to automatically create destination (`io.BytesIO`) use default value of destination and handle result of this method.

Parameters

- **file** – file_id or Downloadable object
- **destination** – Filename, file path or instance of `io.IOBase`. For e.g. `io.BytesIO`, defaults to None
- **timeout** – Total timeout in seconds, defaults to 30
- **chunk_size** – File chunks size, defaults to 64 kb
- **seek** – Go to start of file when downloading is finished. Used only for destination with `typing.BinaryIO` type, defaults to True

2.3.2 Client session

Client sessions is used for interacting with API server.

Use Custom API server

For example, if you want to use self-hosted API server:

```
session = AiohttpSession(
    api=TelegramAPIServer.from_base('http://localhost:8082')
)
bot = Bot(..., session=session)
```

class `aiogram.client.telegram.`**TelegramAPIServer**(*base: str*, *file: str*, *is_local: bool = False*, *wrap_local_file: ~aiogram.client.telegram.FilesPathWrapper = <aiogram.client.telegram.BareFilesPathWrapper object>*)

Base config for API Endpoints

api_url(*token: str*, *method: str*) → str

Generate URL for API methods

Parameters

- **token** – Bot token
- **method** – API method name (case insensitive)

Returns
URL

base: str

Base URL

file: str

Files URL

file_url(*token: str*, *path: str*) → str

Generate URL for downloading files

Parameters

- **token** – Bot token

- **path** – file path

Returns
URL

classmethod from_base(*base: str*, ***kwargs: Any*) → *TelegramAPIServer*

Use this method to auto-generate TelegramAPIServer instance from base URL

Parameters
base – Base URL

Returns
instance of `TelegramAPIServer`

is_local: bool = False

Mark this server is in local mode.

wrap_local_file: FilesPathWrapper = <aiogram.client.telegram.BareFilesPathWrapper object>

Callback to wrap files path in local mode

Base

Abstract session for all client sessions

class aiogram.client.session.base.**BaseSession**(*api: ~aiogram.client.telegram.TelegramAPIServer = TelegramAPIServer(base='https://api.telegram.org/bot{token}/{method}', file='https://api.telegram.org/file/bot{token}/{path}', is_local=False, wrap_local_file=<aiogram.client.telegram.BareFilesPathWrapper object>), json_loads: ~typing.Callable[[...], ~typing.Any] = <function loads>, json_dumps: ~typing.Callable[[...], str] = <function dumps>, timeout: float = 60.0*)

This is base class for all HTTP sessions in aiogram.

If you want to create your own session, you must inherit from this class.

check_response(*bot: Bot*, *method: TelegramMethod[TelegramType]*, *status_code: int*, *content: str*) → Response[TelegramType]

Check response status

abstract async close() → None

Close client session

abstract async make_request(*bot: Bot*, *method: TelegramMethod[TelegramType]*, *timeout: int | None = None*) → TelegramType

Make request to Telegram Bot API

Parameters

- **bot** – Bot instance
- **method** – Method instance
- **timeout** – Request timeout

Returns

Raises
`TelegramApiError` –

`prepare_value`(*value: Any*, *bot: Bot*, *files: Dict[str, Any]*, *_dumps_json: bool = True*) → Any

Prepare value before send

`abstract async` **`stream_content`**(*url: str*, *headers: Dict[str, Any] | None = None*, *timeout: int = 30*, *chunk_size: int = 65536*, *raise_for_status: bool = True*) → AsyncGenerator[bytes, None]

Stream reader

aiohttp

AiohttpSession represents a wrapper-class around *ClientSession* from aiohttp

Currently *AiohttpSession* is a default session used in *aiogram.Bot*

`class aiogram.client.session.aiohttp.`**`AiohttpSession`**(*proxy: Iterable[str | Tuple[str, BasicAuth]] | str | Tuple[str, BasicAuth] | None = None*, *limit: int = 100*, ***kwargs: Any*)

Usage example

```python
from aiogram import Bot
from aiogram.client.session.aiohttp import AiohttpSession

session = AiohttpSession()
bot = Bot('42:token', session=session)
```

Proxy requests in AiohttpSession

In order to use AiohttpSession with proxy connector you have to install aiohttp-socks

Binding session to bot:

```python
from aiogram import Bot
from aiogram.client.session.aiohttp import AiohttpSession

session = AiohttpSession(proxy="protocol://host:port/")
bot = Bot(token="bot token", session=session)
```

Note

Only following protocols are supported: http(tunneling), socks4(a), socks5 as aiohttp_socks documentation claims.

Authorization

Proxy authorization credentials can be specified in proxy URL or come as an instance of `aiohttp.BasicAuth` containing login and password.

Consider examples:

```
from aiohttp import BasicAuth
from aiogram.client.session.aiohttp import AiohttpSession

auth = BasicAuth(login="user", password="password")
session = AiohttpSession(proxy=("protocol://host:port", auth))
```

or simply include your basic auth credential in URL

```
session = AiohttpSession(proxy="protocol://user:password@host:port")
```

Note

Aiogram prefers *BasicAuth* over username and password in URL, so if proxy URL contains login and password and *BasicAuth* object is passed at the same time aiogram will use login and password from *BasicAuth* instance.

Proxy chains

Since aiohttp-socks supports proxy chains, you're able to use them in aiogram

Example of chain proxies:

```
from aiohttp import BasicAuth
from aiogram.client.session.aiohttp import AiohttpSession

auth = BasicAuth(login="user", password="password")
session = AiohttpSession(
    proxy={
        "protocol0://host0:port0",
        "protocol1://user:password@host1:port1",
        ("protocol2://host2:port2", auth),
    }  # can be any iterable if not set
)
```

Client session middlewares

In some cases you may want to add some middlewares to the client session to customize the behavior of the client.

Some useful cases that is:

- Log the outgoing requests
- Customize the request parameters
- Handle rate limiting errors and retry the request
- others …

So, you can do it using client session middlewares. A client session middleware is a function (or callable class) that receives the request and the next middleware to call. The middleware can modify the request and then call the next middleware to continue the request processing.

How to register client session middleware?

Register using register method

```
bot.session.middleware(RequestLogging(ignore_methods=[GetUpdates]))
```

Register using decorator

```
@bot.session.middleware()
async def my_middleware(
    make_request: NextRequestMiddlewareType[TelegramType],
    bot: "Bot",
    method: TelegramMethod[TelegramType],
) -> Response[TelegramType]:
    # do something with request
    return await make_request(bot, method)
```

Example

Class based session middleware

```
class RequestLogging(BaseRequestMiddleware):
    def __init__(self, ignore_methods: Optional[List[Type[TelegramMethod[Any]]]] = None):
        """
        Middleware for logging outgoing requests

        :param ignore_methods: methods to ignore in logging middleware
        """
        self.ignore_methods = ignore_methods if ignore_methods else []

    async def __call__(
        self,
        make_request: NextRequestMiddlewareType[TelegramType],
        bot: "Bot",
        method: TelegramMethod[TelegramType],
    ) -> Response[TelegramType]:
        if type(method) not in self.ignore_methods:
            loggers.middlewares.info(
                "Make request with method=%r by bot id=%d",
                type(method).__name__,
                bot.id,
            )
        return await make_request(bot, method)
```

Note

this middleware is already implemented inside aiogram, so, if you want to use it you can just import it `from aiogram.client.session.middlewares.request_logging import RequestLogging`

Function based session middleware

```python
async def __call__(
    self,
    make_request: NextRequestMiddlewareType[TelegramType],
    bot: "Bot",
    method: TelegramMethod[TelegramType],
) -> Response[TelegramType]:
    try:
        # do something with request
        return await make_request(bot, method)
    finally:
        # do something after request
```

2.3.3 Types

Here is list of all available API types:

Available types

Animation

`class aiogram.types.animation.`**`Animation`**(*, *file_id: str*, *file_unique_id: str*, *width: int*, *height: int*, *duration: int*, *thumbnail: PhotoSize | None = None*, *file_name: str | None = None*, *mime_type: str | None = None*, *file_size: int | None = None*, ***extra_data: Any*)

This object represents an animation file (GIF or H.264/MPEG-4 AVC video without sound).

Source: https://core.telegram.org/bots/api#animation

`file_id: str`

Identifier for this file, which can be used to download or reuse the file

`file_unique_id: str`

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

`width: int`

Video width as defined by the sender

`height: int`

Video height as defined by the sender

`duration: int`

Duration of the video in seconds as defined by the sender

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

thumbnail: *PhotoSize* **| None**

Optional. Animation thumbnail as defined by the sender

file_name: str | None

Optional. Original animation filename as defined by the sender

mime_type: str | None

Optional. MIME type of the file as defined by the sender

file_size: int | None

Optional. File size in bytes. It can be bigger than 2^31 and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this value.

Audio

class aiogram.types.audio.**Audio**(**, file_id: str, file_unique_id: str, duration: int, performer: str | None = None, title: str | None = None, file_name: str | None = None, mime_type: str | None = None, file_size: int | None = None, thumbnail:* PhotoSize *| None = None, **extra_data: Any*)

This object represents an audio file to be treated as music by the Telegram clients.

Source: https://core.telegram.org/bots/api#audio

file_id: str

Identifier for this file, which can be used to download or reuse the file

file_unique_id: str

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

duration: int

Duration of the audio in seconds as defined by the sender

performer: str | None

Optional. Performer of the audio as defined by the sender or by audio tags

title: str | None

Optional. Title of the audio as defined by the sender or by audio tags

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_name: str | None

Optional. Original filename as defined by the sender

mime_type: str | None

Optional. MIME type of the file as defined by the sender

file_size: int | None

Optional. File size in bytes. It can be bigger than 2^31 and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this value.

thumbnail: *PhotoSize* | None

Optional. Thumbnail of the album cover to which the music file belongs

BackgroundFill

class aiogram.types.background_fill.**BackgroundFill**(***extra_data: Any*)

This object describes the way a background is filled based on the selected colors. Currently, it can be one of

- *aiogram.types.background_fill_solid.BackgroundFillSolid*
- *aiogram.types.background_fill_gradient.BackgroundFillGradient*
- *aiogram.types.background_fill_freeform_gradient.BackgroundFillFreeformGradient*

Source: https://core.telegram.org/bots/api#backgroundfill

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BackgroundFillFreeformGradient

class aiogram.types.background_fill_freeform_gradient.**BackgroundFillFreeformGradient**(**, type: Literal['freeform_gradient = 'freeform_gradient', colors: List[int], **extra_data: Any*)

The background is a freeform gradient that rotates after every message in the chat.

Source: https://core.telegram.org/bots/api#backgroundfillfreeformgradient

type: Literal['freeform_gradient']

Type of the background fill, always 'freeform_gradient'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

colors: List[int]

A list of the 3 or 4 base colors that are used to generate the freeform gradient in the RGB24 format

BackgroundFillGradient

class aiogram.types.background_fill_gradient.**BackgroundFillGradient**(**, type: Literal['gradient'] = 'gradient', top_color: int, bottom_color: int, rotation_angle: int, **extra_data: Any*)

The background is a gradient fill.

Source: https://core.telegram.org/bots/api#backgroundfillgradient

type: Literal['gradient']

Type of the background fill, always 'gradient'

top_color: int

Top color of the gradient in the RGB24 format

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

bottom_color: int

Bottom color of the gradient in the RGB24 format

rotation_angle: int

Clockwise rotation angle of the background fill in degrees; 0-359

BackgroundFillSolid

class aiogram.types.background_fill_solid.**BackgroundFillSolid**(**, type: Literal['solid'] = 'solid', color: int, **extra_data: Any*)

The background is filled using the selected color.

Source: https://core.telegram.org/bots/api#backgroundfillsolid

type: Literal['solid']

Type of the background fill, always 'solid'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

color: int

The color of the background fill in the RGB24 format

BackgroundType

class `aiogram.types.background_type.`**`BackgroundType`**(***extra_data: Any*)

This object describes the type of a background. Currently, it can be one of

- *`aiogram.types.background_type_fill.BackgroundTypeFill`*
- *`aiogram.types.background_type_wallpaper.BackgroundTypeWallpaper`*
- *`aiogram.types.background_type_pattern.BackgroundTypePattern`*
- *`aiogram.types.background_type_chat_theme.BackgroundTypeChatTheme`*

Source: https://core.telegram.org/bots/api#backgroundtype

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BackgroundTypeChatTheme

class `aiogram.types.background_type_chat_theme.`**`BackgroundTypeChatTheme`**(**, type: Literal['chat_theme'] = 'chat_theme', theme_name: str, **extra_data: Any*)

The background is taken directly from a built-in chat theme.

Source: https://core.telegram.org/bots/api#backgroundtypechattheme

`type: Literal['chat_theme']`

Type of the background, always 'chat_theme'

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`theme_name: str`

Name of the chat theme, which is usually an emoji

BackgroundTypeFill

class `aiogram.types.background_type_fill.`**`BackgroundTypeFill`**(**, type: Literal['fill'] = 'fill', fill: BackgroundFillSolid | BackgroundFillGradient | BackgroundFillFreeformGradient, dark_theme_dimming: int, **extra_data: Any*)

The background is automatically filled based on the selected colors.

Source: https://core.telegram.org/bots/api#backgroundtypefill

type: Literal['fill']

Type of the background, always 'fill'

fill: *BackgroundFillSolid* **|** *BackgroundFillGradient* **|** *BackgroundFillFreeformGradient*

The background fill

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

dark_theme_dimming: int

Dimming of the background in dark themes, as a percentage; 0-100

BackgroundTypePattern

class aiogram.types.background_type_pattern.**BackgroundTypePattern**(**, type: Literal['pattern'] = 'pattern', document: Document, fill: BackgroundFillSolid | BackgroundFillGradient | BackgroundFillFreeformGradient, intensity: int, is_inverted: bool | None = None, is_moving: bool | None = None, **extra_data: Any*)

The background is a PNG or TGV (gzipped subset of SVG with MIME type 'application/x-tgwallpattern') pattern to be combined with the background fill chosen by the user.

Source: https://core.telegram.org/bots/api#backgroundtypepattern

type: Literal['pattern']

Type of the background, always 'pattern'

document: *Document*

Document with the pattern

fill: *BackgroundFillSolid* **|** *BackgroundFillGradient* **|** *BackgroundFillFreeformGradient*

The background fill that is combined with the pattern

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

intensity: int

Intensity of the pattern when it is shown above the filled background; 0-100

is_inverted: bool | None

Optional. `True`, if the background fill must be applied only to the pattern itself. All other pixels are black in this case. For dark themes only

is_moving: bool | None

Optional. `True`, if the background moves slightly when the device is tilted

BackgroundTypeWallpaper

class aiogram.types.background_type_wallpaper.**BackgroundTypeWallpaper**(**, type: Literal['wallpaper'] = 'wallpaper', document: Document, dark_theme_dimming: int, is_blurred: bool | None = None, is_moving: bool | None = None, **extra_data: Any*)

The background is a wallpaper in the JPEG format.

Source: https://core.telegram.org/bots/api#backgroundtypewallpaper

type: Literal['wallpaper']

Type of the background, always 'wallpaper'

document: *Document*

Document with the wallpaper

dark_theme_dimming: int

Dimming of the background in dark themes, as a percentage; 0-100

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

is_blurred: bool | None

Optional. True, if the wallpaper is downscaled to fit in a 450x450 square and then box-blurred with radius 12

is_moving: bool | None

Optional. True, if the background moves slightly when the device is tilted

Birthdate

class aiogram.types.birthdate.**Birthdate**(**, day: int, month: int, year: int | None = None, **extra_data: Any*)

Describes the birthdate of a user.

Source: https://core.telegram.org/bots/api#birthdate

day: int

Day of the user's birth; 1-31

month: int

Month of the user's birth; 1-12

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

year: int | None

Optional. Year of the user's birth

BotCommand

class aiogram.types.bot_command.**BotCommand**(**, command: str, description: str, **extra_data: Any*)

This object represents a bot command.

Source: https://core.telegram.org/bots/api#botcommand

command: str

Text of the command; 1-32 characters. Can contain only lowercase English letters, digits and underscores.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

description: str

Description of the command; 1-256 characters.

BotCommandScope

class aiogram.types.bot_command_scope.**BotCommandScope**(***extra_data: Any*)

This object represents the scope to which bot commands are applied. Currently, the following 7 scopes are supported:

- *aiogram.types.bot_command_scope_default.BotCommandScopeDefault*
- *aiogram.types.bot_command_scope_all_private_chats.BotCommandScopeAllPrivateChats*
- *aiogram.types.bot_command_scope_all_group_chats.BotCommandScopeAllGroupChats*
- *aiogram.types.bot_command_scope_all_chat_administrators.BotCommandScopeAllChatAdministrators*
- *aiogram.types.bot_command_scope_chat.BotCommandScopeChat*
- *aiogram.types.bot_command_scope_chat_administrators.BotCommandScopeChatAdministrators*
- *aiogram.types.bot_command_scope_chat_member.BotCommandScopeChatMember*

Source: https://core.telegram.org/bots/api#botcommandscope

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BotCommandScopeAllChatAdministrators

class aiogram.types.bot_command_scope_all_chat_administrators.**BotCommandScopeAllChatAdministrators**(*, *type: Literal[B = BotCommandScope **extra_d Any*)

Represents the scope of bot commands, covering all group and supergroup chat administrators.

Source: https://core.telegram.org/bots/api#botcommandscopeallchatadministrators

type: Literal[BotCommandScopeType.ALL_CHAT_ADMINISTRATORS]

Scope type, must be *all_chat_administrators*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BotCommandScopeAllGroupChats

class aiogram.types.bot_command_scope_all_group_chats.**BotCommandScopeAllGroupChats**(*, *type: Literal[BotCommandScopeTy = BotCommandScopeType.ALL_GROUP_ **extra_data: Any*)

Represents the scope of bot commands, covering all group and supergroup chats.

Source: https://core.telegram.org/bots/api#botcommandscopeallgroupchats

type: Literal[BotCommandScopeType.ALL_GROUP_CHATS]

Scope type, must be *all_group_chats*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BotCommandScopeAllPrivateChats

class aiogram.types.bot_command_scope_all_private_chats.**BotCommandScopeAllPrivateChats**(*, *type: Literal[BotCommandSc* *= BotCommandScopeType.ALL_PRI* ***extra_data: Any*)

Represents the scope of bot commands, covering all private chats.

Source: https://core.telegram.org/bots/api#botcommandscopeallprivatechats

type: Literal[BotCommandScopeType.ALL_PRIVATE_CHATS]

Scope type, must be *all_private_chats*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BotCommandScopeChat

class aiogram.types.bot_command_scope_chat.**BotCommandScopeChat**(*, *type: Literal[BotCommandScopeType.CHAT] = BotCommandScopeType.CHAT*, *chat_id: int | str*, ***extra_data: Any*)

Represents the scope of bot commands, covering a specific chat.

Source: https://core.telegram.org/bots/api#botcommandscopechat

type: Literal[BotCommandScopeType.CHAT]

Scope type, must be *chat*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

BotCommandScopeChatAdministrators

class aiogram.types.bot_command_scope_chat_administrators.**BotCommandScopeChatAdministrators**(**, type: Literal[BotComm = BotCommandScopeType.CH chat_id: int | str, **extra_data: Any*)

Represents the scope of bot commands, covering all administrators of a specific group or supergroup chat.

Source: https://core.telegram.org/bots/api#botcommandscopechatadministrators

type: Literal[BotCommandScopeType.CHAT_ADMINISTRATORS]

Scope type, must be *chat_administrators*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

BotCommandScopeChatMember

class aiogram.types.bot_command_scope_chat_member.**BotCommandScopeChatMember**(**, type: Literal[BotCommandScopeType.CHAT = BotCommandScopeType.CHAT_MEMBER, chat_id: int | str, user_id: int, **extra_data: Any*)

Represents the scope of bot commands, covering a specific member of a group or supergroup chat.

Source: https://core.telegram.org/bots/api#botcommandscopechatmember

type: Literal[BotCommandScopeType.CHAT_MEMBER]

Scope type, must be *chat_member*

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_id: int

Unique identifier of the target user

BotCommandScopeDefault

class aiogram.types.bot_command_scope_default.**BotCommandScopeDefault**(*, *type: Literal[BotCommandScopeType.DEFAULT] = BotCommandScopeType.DEFAULT*, ***extra_data: Any*)

Represents the default scope of bot commands. Default commands are used if no commands with a narrower scope are specified for the user.

Source: https://core.telegram.org/bots/api#botcommandscopedefault

type: Literal[BotCommandScopeType.DEFAULT]

Scope type, must be *default*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BotDescription

class aiogram.types.bot_description.**BotDescription**(*, *description: str*, ***extra_data: Any*)

This object represents the bot's description.

Source: https://core.telegram.org/bots/api#botdescription

description: str

The bot's description

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BotName

class aiogram.types.bot_name.**BotName**(*, *name: str*, ***extra_data: Any*)

This object represents the bot's name.

Source: https://core.telegram.org/bots/api#botname

name: str

The bot's name

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BotShortDescription

class aiogram.types.bot_short_description.**BotShortDescription**(*, *short_description: str*, ***extra_data: Any*)

This object represents the bot's short description.

Source: https://core.telegram.org/bots/api#botshortdescription

short_description: str

The bot's short description

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

BusinessConnection

class aiogram.types.business_connection.**BusinessConnection**(*, *id: str*, *user: User*, *user_chat_id: int*, *date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *can_reply: bool*, *is_enabled: bool*, ***extra_data: Any*)

Describes the connection of the bot with a business account.

Source: https://core.telegram.org/bots/api#businessconnection

id: str

Unique identifier of the business connection

user: *User*

Business account user that created the business connection

user_chat_id: int

Identifier of a private chat with the user who created the business connection. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a 64-bit integer or double-precision float type are safe for storing this identifier.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

date: DateTime

Date the connection was established in Unix time

can_reply: bool

True, if the bot can act on behalf of the business account in chats that were active in the last 24 hours

is_enabled: bool

True, if the connection is active

BusinessIntro

class aiogram.types.business_intro.**BusinessIntro**(**, title: str | None = None, message: str | None = None, sticker: Sticker | None = None, **extra_data: Any*)

Contains information about the start page settings of a Telegram Business account.

Source: https://core.telegram.org/bots/api#businessintro

title: str | None

Optional. Title text of the business intro

message: str | None

Optional. Message text of the business intro

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sticker: *Sticker* | None

Optional. Sticker of the business intro

BusinessLocation

class aiogram.types.business_location.**BusinessLocation**(**, address: str, location: Location | None = None, **extra_data: Any*)

Contains information about the location of a Telegram Business account.

Source: https://core.telegram.org/bots/api#businesslocation

address: str

Address of the business

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

location: *Location* | None

Optional. Location of the business

BusinessMessagesDeleted

class aiogram.types.business_messages_deleted.**BusinessMessagesDeleted**(*, *business_connection_id: str*, *chat: Chat*, *message_ids: List[int]*, ***extra_data: Any*)

This object is received when messages are deleted from a connected business account.

Source: https://core.telegram.org/bots/api#businessmessagesdeleted

business_connection_id: str

Unique identifier of the business connection

chat: *Chat*

Information about a chat in the business account. The bot may not have access to the chat or the corresponding user.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_ids: List[int]

The list of identifiers of deleted messages in the chat of the business account

BusinessOpeningHours

class aiogram.types.business_opening_hours.**BusinessOpeningHours**(*, *time_zone_name: str*, *opening_hours: List[BusinessOpeningHoursInterval]*, ***extra_data: Any*)

Describes the opening hours of a business.

Source: https://core.telegram.org/bots/api#businessopeninghours

time_zone_name: str

Unique name of the time zone for which the opening hours are defined

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

opening_hours: List[*BusinessOpeningHoursInterval*]

List of time intervals describing business opening hours

BusinessOpeningHoursInterval

class aiogram.types.business_opening_hours_interval.**BusinessOpeningHoursInterval**(**, opening_minute: int, closing_minute: int, **extra_data: Any*)

Describes an interval of time during which a business is open.

Source: https://core.telegram.org/bots/api#businessopeninghoursinterval

opening_minute: int

The minute's sequence number in a week, starting on Monday, marking the start of the time interval during which the business is open; 0 - 7 * 24 * 60

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

closing_minute: int

The minute's sequence number in a week, starting on Monday, marking the end of the time interval during which the business is open; 0 - 8 * 24 * 60

CallbackQuery

class aiogram.types.callback_query.**CallbackQuery**(**, id: str, from_user: User, chat_instance: str, message: Message | InaccessibleMessage | None = None, inline_message_id: str | None = None, data: str | None = None, game_short_name: str | None = None, **extra_data: Any*)

This object represents an incoming callback query from a callback button in an inline keyboard. If the button that originated the query was attached to a message sent by the bot, the field *message* will be present. If the button was attached to a message sent via the bot (in inline mode), the field *inline_message_id* will be present. Exactly one of the fields *data* or *game_short_name* will be present.

NOTE: After the user presses a callback button, Telegram clients will display a progress bar until you call *aiogram.methods.answer_callback_query.AnswerCallbackQuery*. It is, therefore, necessary to react by calling *aiogram.methods.answer_callback_query.AnswerCallbackQuery* even if no notification to the user is needed (e.g., without specifying any of the optional parameters).

Source: https://core.telegram.org/bots/api#callbackquery

id: str

Unique identifier for this query

from_user: *User*

Sender

chat_instance: str

Global identifier, uniquely corresponding to the chat to which the message with the callback button was sent. Useful for high scores in `aiogram.methods.games.Games`.

message: *Message* | *InaccessibleMessage* | None

Optional. Message sent by the bot with the callback button that originated the query

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

inline_message_id: str | None

Optional. Identifier of the message sent via the bot in inline mode, that originated the query.

data: str | None

Optional. Data associated with the callback button. Be aware that the message originated the query can contain no callback buttons with this data.

game_short_name: str | None

Optional. Short name of a Game to be returned, serves as the unique identifier for the game

answer(*text: str | None = None*, *show_alert: bool | None = None*, *url: str | None = None*, *cache_time: int | None = None*, ***kwargs: Any*) → *AnswerCallbackQuery*

Shortcut for method *aiogram.methods.answer_callback_query.AnswerCallbackQuery* will automatically fill method attributes:

- `callback_query_id`

Use this method to send answers to callback queries sent from inline keyboards. The answer will be displayed to the user as a notification at the top of the chat screen or as an alert. On success, `True` is returned.

Alternatively, the user can be redirected to the specified Game URL. For this option to work, you must first create a game for your bot via @BotFather and accept the terms. Otherwise, you may use links like `t.me/your_bot?start=XXXX` that open your bot with a parameter.

Source: https://core.telegram.org/bots/api#answercallbackquery

Parameters

- **text** – Text of the notification. If not specified, nothing will be shown to the user, 0-200 characters
- **show_alert** – If `True`, an alert will be shown by the client instead of a notification at the top of the chat screen. Defaults to *false*.
- **url** – URL that will be opened by the user's client. If you have created a *aiogram.types.game.Game* and accepted the conditions via @BotFather, specify the URL that opens your game - note that this will only work if the query comes from a https://core.telegram.org/bots/api#inlinekeyboardbutton *callback_game* button.
- **cache_time** – The maximum amount of time in seconds that the result of the callback query may be cached client-side. Telegram apps will support caching starting in version 3.14. Defaults to 0.

Returns

instance of method *aiogram.methods.answer_callback_query.AnswerCallbackQuery*

Chat

class aiogram.types.chat.**Chat**(**, id: int, type: str, title: str | None = None, username: str | None = None, first_name: str | None = None, last_name: str | None = None, is_forum: bool | None = None, accent_color_id: int | None = None, active_usernames: List[str] | None = None, available_reactions: List[*ReactionTypeEmoji | ReactionTypeCustomEmoji | ReactionTypePaid*] | None = None, background_custom_emoji_id: str | None = None, bio: str | None = None, birthdate:* Birthdate *| None = None, business_intro:* BusinessIntro *| None = None, business_location:* BusinessLocation *| None = None, business_opening_hours:* BusinessOpeningHours *| None = None, can_set_sticker_set: bool | None = None, custom_emoji_sticker_set_name: str | None = None, description: str | None = None, emoji_status_custom_emoji_id: str | None = None, emoji_status_expiration_date: _datetime_serializer, return_type=int, when_used=unless - none)] | None = None, has_aggressive_anti_spam_enabled: bool | None = None, has_hidden_members: bool | None = None, has_private_forwards: bool | None = None, has_protected_content: bool | None = None, has_restricted_voice_and_video_messages: bool | None = None, has_visible_history: bool | None = None, invite_link: str | None = None, join_by_request: bool | None = None, join_to_send_messages: bool | None = None, linked_chat_id: int | None = None, location:* ChatLocation *| None = None, message_auto_delete_time: int | None = None, permissions:* ChatPermissions *| None = None, personal_chat:* Chat *| None = None, photo:* ChatPhoto *| None = None, pinned_message:* Message *| None = None, profile_accent_color_id: int | None = None, profile_background_custom_emoji_id: str | None = None, slow_mode_delay: int | None = None, sticker_set_name: str | None = None, unrestrict_boost_count: int | None = None, **extra_data: Any*)

This object represents a chat.

Source: https://core.telegram.org/bots/api#chat

id: int

Unique identifier for this chat. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this identifier.

type: str

Type of the chat, can be either 'private', 'group', 'supergroup' or 'channel'

title: str | None

Optional. Title, for supergroups, channels and group chats

username: str | None

Optional. Username, for private chats, supergroups and channels if available

first_name: str | None

Optional. First name of the other party in a private chat

last_name: str | None

Optional. Last name of the other party in a private chat

is_forum: bool | None

Optional. `True`, if the supergroup chat is a forum (has topics enabled)

accent_color_id: int | None

Optional. Identifier of the accent color for the chat name and backgrounds of the chat photo, reply header, and link preview. See accent colors for more details. Returned only in `aiogram.methods.get_chat.GetChat`. Always returned in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

active_usernames: List[str] | None

Optional. If non-empty, the list of all active chat usernames; for private chats, supergroups and channels. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

available_reactions: List[*ReactionTypeEmoji* | *ReactionTypeCustomEmoji* | *ReactionTypePaid*] | None

Optional. List of available reactions allowed in the chat. If omitted, then all emoji reactions are allowed. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

background_custom_emoji_id: str | None

Optional. Custom emoji identifier of emoji chosen by the chat for the reply header and link preview background. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

bio: str | None

Optional. Bio of the other party in a private chat. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

birthdate: *Birthdate* | None

Optional. For private chats, the date of birth of the user. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

business_intro: *BusinessIntro* | None

Optional. For private chats with business accounts, the intro of the business. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

business_location: *BusinessLocation* | None

Optional. For private chats with business accounts, the location of the business. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

business_opening_hours: *BusinessOpeningHours* | None

Optional. For private chats with business accounts, the opening hours of the business. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

can_set_sticker_set: bool | None

Optional. `True`, if the bot can change the group sticker set. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

custom_emoji_sticker_set_name: str | None

Optional. For supergroups, the name of the group's custom emoji sticker set. Custom emoji from this set can be used by all users and bots in the group. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

description: str | None

Optional. Description, for groups, supergroups and channel chats. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

emoji_status_custom_emoji_id: str | None

Optional. Custom emoji identifier of the emoji status of the chat or the other party in a private chat. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

emoji_status_expiration_date: DateTime | None

Optional. Expiration date of the emoji status of the chat or the other party in a private chat, in Unix time, if any. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

has_aggressive_anti_spam_enabled: bool | None

Optional. `True`, if aggressive anti-spam checks are enabled in the supergroup. The field is only available to chat administrators. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

has_hidden_members: bool | None

Optional. `True`, if non-administrators can only get the list of bots and administrators in the chat. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

has_private_forwards: bool | None

Optional. `True`, if privacy settings of the other party in the private chat allows to use `tg://user?id=<user_id>` links only in chats with the user. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

has_protected_content: bool | None

Optional. `True`, if messages from the chat can't be forwarded to other chats. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

has_restricted_voice_and_video_messages: bool | None

Optional. `True`, if the privacy settings of the other party restrict sending voice and video note messages in the private chat. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

has_visible_history: bool | None

Optional. `True`, if new chat members will have access to old messages; available only to chat administrators. Returned only in *aiogram.methods.get_chat.GetChat*.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

invite_link: str | None

Optional. Primary invite link, for groups, supergroups and channel chats. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

join_by_request: bool | None

Optional. `True`, if all users directly joining the supergroup need to be approved by supergroup administrators. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

join_to_send_messages: bool | None

Optional. `True`, if users need to join the supergroup before they can send messages. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

linked_chat_id: int | None

Optional. Unique identifier for the linked chat, i.e. the discussion group identifier for a channel and vice versa; for supergroups and channel chats. This identifier may be greater than 32 bits and some programming languages may have difficulty/silent defects in interpreting it. But it is smaller than 52 bits, so a signed 64 bit integer or double-precision float type are safe for storing this identifier. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

location: *ChatLocation* | None

Optional. For supergroups, the location to which the supergroup is connected. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

message_auto_delete_time: int | None

Optional. The time after which all messages sent to the chat will be automatically deleted; in seconds. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

permissions: *ChatPermissions* | None

Optional. Default chat member permissions, for groups and supergroups. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

personal_chat: *Chat* | None

Optional. For private chats, the personal channel of the user. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

photo: *ChatPhoto* | None

Optional. Chat photo. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

pinned_message: *Message* **| None**

Optional. The most recent pinned message (by sending date). Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

profile_accent_color_id: int | None

Optional. Identifier of the accent color for the chat's profile background. See profile accent colors for more details. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

profile_background_custom_emoji_id: str | None

Optional. Custom emoji identifier of the emoji chosen by the chat for its profile background. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

slow_mode_delay: int | None

Optional. For supergroups, the minimum allowed delay between consecutive messages sent by each unprivileged user; in seconds. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

sticker_set_name: str | None

Optional. For supergroups, name of group sticker set. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

unrestrict_boost_count: int | None

Optional. For supergroups, the minimum number of boosts that a non-administrator user needs to add in order to ignore slow mode and chat permissions. Returned only in `aiogram.methods.get_chat.GetChat`.

Deprecated since version API:7.3: https://core.telegram.org/bots/api-changelog#may-6-2024

property shifted_id: int

Returns shifted chat ID (positive and without "-100" prefix). Mostly used for private links like t.me/c/chat_id/message_id

Currently supergroup/channel IDs have 10-digit ID after "-100" prefix removed. However, these IDs might become 11-digit in future. So, first we remove "-100" prefix and count remaining number length. Then we multiple -1 * 10 ^ (number_length + 2) Finally, self.id is substracted from that number

property full_name: str

Get full name of the Chat.

For private chat it is first_name + last_name. For other chat types it is title.

ban_sender_chat(*sender_chat_id: int*, ***kwargs: Any*) → *BanChatSenderChat*

Shortcut for method `aiogram.methods.ban_chat_sender_chat.BanChatSenderChat` will automatically fill method attributes:

- `chat_id`

Use this method to ban a channel chat in a supergroup or a channel. Until the chat is unbanned, the owner of the banned chat won't be able to send messages on behalf of **any of their channels**. The bot must be an administrator in the supergroup or channel for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#banchatsenderchat

Parameters
`sender_chat_id` – Unique identifier of the target sender chat

Returns
instance of method *`aiogram.methods.ban_chat_sender_chat.BanChatSenderChat`*

`unban_sender_chat`(*sender_chat_id: int*, ***kwargs: Any*) → *UnbanChatSenderChat*

Shortcut for method *`aiogram.methods.unban_chat_sender_chat.UnbanChatSenderChat`* will automatically fill method attributes:

- `chat_id`

Use this method to unban a previously banned channel chat in a supergroup or channel. The bot must be an administrator for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unbanchatsenderchat

Parameters
`sender_chat_id` – Unique identifier of the target sender chat

Returns
instance of method *`aiogram.methods.unban_chat_sender_chat.UnbanChatSenderChat`*

`get_administrators`(***kwargs: Any*) → *GetChatAdministrators*

Shortcut for method *`aiogram.methods.get_chat_administrators.GetChatAdministrators`* will automatically fill method attributes:

- `chat_id`

Use this method to get a list of administrators in a chat, which aren't bots. Returns an Array of *`aiogram.types.chat_member.ChatMember`* objects.

Source: https://core.telegram.org/bots/api#getchatadministrators

Returns
instance of method *`aiogram.methods.get_chat_administrators.GetChatAdministrators`*

`delete_message`(*message_id: int*, ***kwargs: Any*) → *DeleteMessage*

Shortcut for method *`aiogram.methods.delete_message.DeleteMessage`* will automatically fill method attributes:

- `chat_id`

Use this method to delete a message, including service messages, with the following limitations:

- A message can only be deleted if it was sent less than 48 hours ago.
- Service messages about a supergroup, channel, or forum topic creation can't be deleted.
- A dice message in a private chat can only be deleted if it was sent more than 24 hours ago.
- Bots can delete outgoing messages in private chats, groups, and supergroups.
- Bots can delete incoming messages in private chats.
- Bots granted *can_post_messages* permissions can delete outgoing messages in channels.

- If the bot is an administrator of a group, it can delete any message there.
- If the bot has *can_delete_messages* permission in a supergroup or a channel, it can delete any message there.

Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletemessage

Parameters
`message_id` – Identifier of the message to delete

Returns
instance of method *`aiogram.methods.delete_message.DeleteMessage`*

`revoke_invite_link`(*invite_link: str*, ***kwargs: Any*) → *RevokeChatInviteLink*

Shortcut for method *`aiogram.methods.revoke_chat_invite_link.RevokeChatInviteLink`* will automatically fill method attributes:

- `chat_id`

Use this method to revoke an invite link created by the bot. If the primary link is revoked, a new link is automatically generated. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns the revoked invite link as *`aiogram.types.chat_invite_link.ChatInviteLink`* object.

Source: https://core.telegram.org/bots/api#revokechatinvitelink

Parameters
`invite_link` – The invite link to revoke

Returns
instance of method *`aiogram.methods.revoke_chat_invite_link.RevokeChatInviteLink`*

`edit_invite_link`(*invite_link: str*, *name: str | None = None*, *expire_date: datetime.datetime | datetime.timedelta | int | None = None*, *member_limit: int | None = None*, *creates_join_request: bool | None = None*, ***kwargs: Any*) → *EditChatInviteLink*

Shortcut for method *`aiogram.methods.edit_chat_invite_link.EditChatInviteLink`* will automatically fill method attributes:

- `chat_id`

Use this method to edit a non-primary invite link created by the bot. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns the edited invite link as a *`aiogram.types.chat_invite_link.ChatInviteLink`* object.

Source: https://core.telegram.org/bots/api#editchatinvitelink

Parameters

- **`invite_link`** – The invite link to edit
- **`name`** – Invite link name; 0-32 characters
- **`expire_date`** – Point in time (Unix timestamp) when the link will expire
- **`member_limit`** – The maximum number of users that can be members of the chat simultaneously after joining the chat via this invite link; 1-99999
- **`creates_join_request`** – `True`, if users joining the chat via the link need to be approved by chat administrators. If `True`, *member_limit* can't be specified

Returns
instance of method `aiogram.methods.edit_chat_invite_link.EditChatInviteLink`

create_invite_link(*name: str | None = None*, *expire_date: datetime.datetime | datetime.timedelta | int | None = None*, *member_limit: int | None = None*, *creates_join_request: bool | None = None*, ***kwargs: Any*) → *CreateChatInviteLink*

Shortcut for method `aiogram.methods.create_chat_invite_link.CreateChatInviteLink` will automatically fill method attributes:

- `chat_id`

Use this method to create an additional invite link for a chat. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. The link can be revoked using the method `aiogram.methods.revoke_chat_invite_link.RevokeChatInviteLink`. Returns the new invite link as `aiogram.types.chat_invite_link.ChatInviteLink` object.

Source: https://core.telegram.org/bots/api#createchatinvitelink

Parameters

- **name** – Invite link name; 0-32 characters
- **expire_date** – Point in time (Unix timestamp) when the link will expire
- **member_limit** – The maximum number of users that can be members of the chat simultaneously after joining the chat via this invite link; 1-99999
- **creates_join_request** – `True`, if users joining the chat via the link need to be approved by chat administrators. If `True`, *member_limit* can't be specified

Returns
instance of method `aiogram.methods.create_chat_invite_link.CreateChatInviteLink`

export_invite_link(***kwargs: Any*) → *ExportChatInviteLink*

Shortcut for method `aiogram.methods.export_chat_invite_link.ExportChatInviteLink` will automatically fill method attributes:

- `chat_id`

Use this method to generate a new primary invite link for a chat; any previously generated primary link is revoked. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns the new invite link as *String* on success.

Note: Each administrator in a chat generates their own invite links. Bots can't use invite links generated by other administrators. If you want your bot to work with invite links, it will need to generate its own link using `aiogram.methods.export_chat_invite_link.ExportChatInviteLink` or by calling the `aiogram.methods.get_chat.GetChat` method. If your bot needs to generate a new primary invite link replacing its previous one, use `aiogram.methods.export_chat_invite_link.ExportChatInviteLink` again.

Source: https://core.telegram.org/bots/api#exportchatinvitelink

Returns
instance of method `aiogram.methods.export_chat_invite_link.ExportChatInviteLink`

do(*action: str*, *business_connection_id: str | None = None*, *message_thread_id: int | None = None*, ***kwargs: Any*) → *SendChatAction*

Shortcut for method `aiogram.methods.send_chat_action.SendChatAction` will automatically fill method attributes:

- chat_id

Use this method when you need to tell the user that something is happening on the bot's side. The status is set for 5 seconds or less (when a message arrives from your bot, Telegram clients clear its typing status). Returns True on success.

> Example: The ImageBot needs some time to process a request and upload the image. Instead of sending a text message along the lines of 'Retrieving image, please wait…', the bot may use *aiogram.methods.send_chat_action.SendChatAction* with *action = upload_photo*. The user will see a 'sending photo' status for the bot.

We only recommend using this method when a response from the bot will take a **noticeable** amount of time to arrive.

Source: https://core.telegram.org/bots/api#sendchataction

Parameters

- **action** – Type of action to broadcast. Choose one, depending on what the user is about to receive: *typing* for text messages, *upload_photo* for photos, *record_video* or *upload_video* for videos, *record_voice* or *upload_voice* for voice notes, *upload_document* for general files, *choose_sticker* for stickers, *find_location* for location data, *record_video_note* or *upload_video_note* for video notes.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the action will be sent
- **message_thread_id** – Unique identifier for the target message thread; for supergroups only

Returns
instance of method *aiogram.methods.send_chat_action.SendChatAction*

delete_sticker_set(***kwargs: Any*) → *DeleteChatStickerSet*

Shortcut for method *aiogram.methods.delete_chat_sticker_set.DeleteChatStickerSet* will automatically fill method attributes:

- chat_id

Use this method to delete a group sticker set from a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Use the field *can_set_sticker_set* optionally returned in *aiogram.methods.get_chat.GetChat* requests to check if the bot can use this method. Returns True on success.

Source: https://core.telegram.org/bots/api#deletechatstickerset

Returns
instance of method *aiogram.methods.delete_chat_sticker_set.DeleteChatStickerSet*

set_sticker_set(*sticker_set_name: str*, ***kwargs: Any*) → *SetChatStickerSet*

Shortcut for method *aiogram.methods.set_chat_sticker_set.SetChatStickerSet* will automatically fill method attributes:

- chat_id

Use this method to set a new group sticker set for a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Use the field *can_set_sticker_set* optionally returned in *aiogram.methods.get_chat.GetChat* requests to check if the bot can use this method. Returns True on success.

Source: https://core.telegram.org/bots/api#setchatstickerset

Parameters
`sticker_set_name` – Name of the sticker set to be set as the group sticker set

Returns
instance of method `aiogram.methods.set_chat_sticker_set.SetChatStickerSet`

`get_member`(*user_id: int*, ***kwargs: Any*) → *GetChatMember*

Shortcut for method `aiogram.methods.get_chat_member.GetChatMember` will automatically fill method attributes:

- `chat_id`

Use this method to get information about a member of a chat. The method is only guaranteed to work for other users if the bot is an administrator in the chat. Returns a `aiogram.types.chat_member.ChatMember` object on success.

Source: https://core.telegram.org/bots/api#getchatmember

Parameters
`user_id` – Unique identifier of the target user

Returns
instance of method `aiogram.methods.get_chat_member.GetChatMember`

`get_member_count`(***kwargs: Any*) → *GetChatMemberCount*

Shortcut for method `aiogram.methods.get_chat_member_count.GetChatMemberCount` will automatically fill method attributes:

- `chat_id`

Use this method to get the number of members in a chat. Returns *Int* on success.

Source: https://core.telegram.org/bots/api#getchatmembercount

Returns
instance of method `aiogram.methods.get_chat_member_count.GetChatMemberCount`

`leave`(***kwargs: Any*) → *LeaveChat*

Shortcut for method `aiogram.methods.leave_chat.LeaveChat` will automatically fill method attributes:

- `chat_id`

Use this method for your bot to leave a group, supergroup or channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#leavechat

Returns
instance of method `aiogram.methods.leave_chat.LeaveChat`

`unpin_all_messages`(***kwargs: Any*) → *UnpinAllChatMessages*

Shortcut for method `aiogram.methods.unpin_all_chat_messages.UnpinAllChatMessages` will automatically fill method attributes:

- `chat_id`

Use this method to clear the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unpinallchatmessages

Returns
instance of method `aiogram.methods.unpin_all_chat_messages.UnpinAllChatMessages`

unpin_message(*business_connection_id: str | None = None*, *message_id: int | None = None*, ***kwargs: Any*) → *UnpinChatMessage*

Shortcut for method `aiogram.methods.unpin_chat_message.UnpinChatMessage` will automatically fill method attributes:

- `chat_id`

Use this method to remove a message from the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unpinchatmessage

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be unpinned
- **message_id** – Identifier of the message to unpin. Required if *business_connection_id* is specified. If not specified, the most recent pinned message (by sending date) will be unpinned.

Returns
instance of method `aiogram.methods.unpin_chat_message.UnpinChatMessage`

pin_message(*message_id: int*, *business_connection_id: str | None = None*, *disable_notification: bool | None = None*, ***kwargs: Any*) → *PinChatMessage*

Shortcut for method `aiogram.methods.pin_chat_message.PinChatMessage` will automatically fill method attributes:

- `chat_id`

Use this method to add a message to the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#pinchatmessage

Parameters

- **message_id** – Identifier of a message to pin
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be pinned
- **disable_notification** – Pass `True` if it is not necessary to send a notification to all chat members about the new pinned message. Notifications are always disabled in channels and private chats.

Returns
instance of method `aiogram.methods.pin_chat_message.PinChatMessage`

set_administrator_custom_title(*user_id: int*, *custom_title: str*, ***kwargs: Any*) → *SetChatAdministratorCustomTitle*

Shortcut for method `aiogram.methods.set_chat_administrator_custom_title.SetChatAdministratorCustomTitle` will automatically fill method attributes:

- `chat_id`

Use this method to set a custom title for an administrator in a supergroup promoted by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatadministratorcustomtitle

Parameters

- **user_id** – Unique identifier of the target user
- **custom_title** – New custom title for the administrator; 0-16 characters, emoji are not allowed

Returns
instance of method *aiogram.methods.set_chat_administrator_custom_title.SetChatAdministratorCustomTitle*

set_permissions(*permissions:* ChatPermissions, *use_independent_chat_permissions: bool | None = None*, ***kwargs: Any*) → *SetChatPermissions*

Shortcut for method *aiogram.methods.set_chat_permissions.SetChatPermissions* will automatically fill method attributes:

- `chat_id`

Use this method to set default chat permissions for all members. The bot must be an administrator in the group or a supergroup for this to work and must have the *can_restrict_members* administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatpermissions

Parameters

- **permissions** – A JSON-serialized object for new default chat permissions
- **use_independent_chat_permissions** – Pass `True` if chat permissions are set independently. Otherwise, the *can_send_other_messages* and *can_add_web_page_previews* permissions will imply the *can_send_messages*, *can_send_audios*, *can_send_documents*, *can_send_photos*, *can_send_videos*, *can_send_video_notes*, and *can_send_voice_notes* permissions; the *can_send_polls* permission will imply the *can_send_messages* permission.

Returns
instance of method *aiogram.methods.set_chat_permissions.SetChatPermissions*

promote(*user_id: int*, *is_anonymous: bool | None = None*, *can_manage_chat: bool | None = None*, *can_delete_messages: bool | None = None*, *can_manage_video_chats: bool | None = None*, *can_restrict_members: bool | None = None*, *can_promote_members: bool | None = None*, *can_change_info: bool | None = None*, *can_invite_users: bool | None = None*, *can_post_stories: bool | None = None*, *can_edit_stories: bool | None = None*, *can_delete_stories: bool | None = None*, *can_post_messages: bool | None = None*, *can_edit_messages: bool | None = None*, *can_pin_messages: bool | None = None*, *can_manage_topics: bool | None = None*, ***kwargs: Any*) → *PromoteChatMember*

Shortcut for method *aiogram.methods.promote_chat_member.PromoteChatMember* will automatically fill method attributes:

- `chat_id`

Use this method to promote or demote a user in a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Pass `False` for all boolean parameters to demote a user. Returns `True` on success.

Source: https://core.telegram.org/bots/api#promotechatmember

Parameters

- **user_id** – Unique identifier of the target user
- **is_anonymous** – Pass `True` if the administrator's presence in the chat is hidden
- **can_manage_chat** – Pass `True` if the administrator can access the chat event log, get boost list, see hidden supergroup and channel members, report spam messages and ignore slow mode. Implied by any other administrator privilege.
- **can_delete_messages** – Pass `True` if the administrator can delete messages of other users
- **can_manage_video_chats** – Pass `True` if the administrator can manage video chats
- **can_restrict_members** – Pass `True` if the administrator can restrict, ban or unban chat members, or access supergroup statistics
- **can_promote_members** – Pass `True` if the administrator can add new administrators with a subset of their own privileges or demote administrators that they have promoted, directly or indirectly (promoted by administrators that were appointed by him)
- **can_change_info** – Pass `True` if the administrator can change chat title, photo and other settings
- **can_invite_users** – Pass `True` if the administrator can invite new users to the chat
- **can_post_stories** – Pass `True` if the administrator can post stories to the chat
- **can_edit_stories** – Pass `True` if the administrator can edit stories posted by other users, post stories to the chat page, pin chat stories, and access the chat's story archive
- **can_delete_stories** – Pass `True` if the administrator can delete stories posted by other users
- **can_post_messages** – Pass `True` if the administrator can post messages in the channel, or access channel statistics; for channels only
- **can_edit_messages** – Pass `True` if the administrator can edit messages of other users and can pin messages; for channels only
- **can_pin_messages** – Pass `True` if the administrator can pin messages; for supergroups only
- **can_manage_topics** – Pass `True` if the user is allowed to create, rename, close, and reopen forum topics; for supergroups only

Returns

instance of method `aiogram.methods.promote_chat_member.PromoteChatMember`

restrict(*user_id: int*, *permissions: ChatPermissions*, *use_independent_chat_permissions: bool | None = None*, *until_date: datetime.datetime | datetime.timedelta | int | None = None*, ***kwargs: Any*) → *RestrictChatMember*

Shortcut for method `aiogram.methods.restrict_chat_member.RestrictChatMember` will automatically fill method attributes:

- `chat_id`

Use this method to restrict a user in a supergroup. The bot must be an administrator in the supergroup for this to work and must have the appropriate administrator rights. Pass `True` for all permissions to lift restrictions from a user. Returns `True` on success.

Source: https://core.telegram.org/bots/api#restrictchatmember

Parameters

- **`user_id`** – Unique identifier of the target user
- **`permissions`** – A JSON-serialized object for new user permissions
- **`use_independent_chat_permissions`** – Pass `True` if chat permissions are set independently. Otherwise, the *can_send_other_messages* and *can_add_web_page_previews* permissions will imply the *can_send_messages*, *can_send_audios*, *can_send_documents*, *can_send_photos*, *can_send_videos*, *can_send_video_notes*, and *can_send_voice_notes* permissions; the *can_send_polls* permission will imply the *can_send_messages* permission.
- **`until_date`** – Date when restrictions will be lifted for the user; Unix time. If user is restricted for more than 366 days or less than 30 seconds from the current time, they are considered to be restricted forever

Returns

instance of method `aiogram.methods.restrict_chat_member.RestrictChatMember`

`unban`(*user_id: int*, *only_if_banned: bool | None = None*, ***kwargs: Any*) → *UnbanChatMember*

Shortcut for method `aiogram.methods.unban_chat_member.UnbanChatMember` will automatically fill method attributes:

- `chat_id`

Use this method to unban a previously banned user in a supergroup or channel. The user will **not** return to the group or channel automatically, but will be able to join via link, etc. The bot must be an administrator for this to work. By default, this method guarantees that after the call the user is not a member of the chat, but will be able to join it. So if the user is a member of the chat they will also be **removed** from the chat. If you don't want this, use the parameter *only_if_banned*. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unbanchatmember

Parameters

- **`user_id`** – Unique identifier of the target user
- **`only_if_banned`** – Do nothing if the user is not banned

Returns

instance of method `aiogram.methods.unban_chat_member.UnbanChatMember`

`ban`(*user_id: int*, *until_date: datetime.datetime | datetime.timedelta | int | None = None*, *revoke_messages: bool | None = None*, ***kwargs: Any*) → *BanChatMember*

Shortcut for method `aiogram.methods.ban_chat_member.BanChatMember` will automatically fill method attributes:

- `chat_id`

Use this method to ban a user in a group, a supergroup or a channel. In the case of supergroups and channels, the user will not be able to return to the chat on their own using invite links, etc., unless unbanned first. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#banchatmember

Parameters

- **`user_id`** – Unique identifier of the target user

- **until_date** – Date when the user will be unbanned; Unix time. If user is banned for more than 366 days or less than 30 seconds from the current time they are considered to be banned forever. Applied for supergroups and channels only.
- **revoke_messages** – Pass `True` to delete all messages from the chat for the user that is being removed. If `False`, the user will be able to see messages in the group that were sent before the user was removed. Always `True` for supergroups and channels.

Returns
instance of method `aiogram.methods.ban_chat_member.BanChatMember`

set_description(*description: str | None = None*, ***kwargs: Any*) → *SetChatDescription*

Shortcut for method `aiogram.methods.set_chat_description.SetChatDescription` will automatically fill method attributes:

- `chat_id`

Use this method to change the description of a group, a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatdescription

Parameters
description – New chat description, 0-255 characters

Returns
instance of method `aiogram.methods.set_chat_description.SetChatDescription`

set_title(*title: str*, ***kwargs: Any*) → *SetChatTitle*

Shortcut for method `aiogram.methods.set_chat_title.SetChatTitle` will automatically fill method attributes:

- `chat_id`

Use this method to change the title of a chat. Titles can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchattitle

Parameters
title – New chat title, 1-128 characters

Returns
instance of method `aiogram.methods.set_chat_title.SetChatTitle`

delete_photo(***kwargs: Any*) → *DeleteChatPhoto*

Shortcut for method `aiogram.methods.delete_chat_photo.DeleteChatPhoto` will automatically fill method attributes:

- `chat_id`

Use this method to delete a chat photo. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletechatphoto

Returns
instance of method `aiogram.methods.delete_chat_photo.DeleteChatPhoto`

set_photo(*photo:* InputFile, ***kwargs: Any*) → *SetChatPhoto*

Shortcut for method *aiogram.methods.set_chat_photo.SetChatPhoto* will automatically fill method attributes:

- chat_id

Use this method to set a new profile photo for the chat. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns True on success.

Source: https://core.telegram.org/bots/api#setchatphoto

Parameters
photo – New chat photo, uploaded using multipart/form-data

Returns
instance of method *aiogram.methods.set_chat_photo.SetChatPhoto*

unpin_all_general_forum_topic_messages(***kwargs: Any*) → *UnpinAllGeneralForumTopicMessages*

Shortcut for method *aiogram.methods.unpin_all_general_forum_topic_messages.UnpinAllGeneralForumTopicMessages* will automatically fill method attributes:

- chat_id

Use this method to clear the list of pinned messages in a General forum topic. The bot must be an administrator in the chat for this to work and must have the *can_pin_messages* administrator right in the supergroup. Returns True on success.

Source: https://core.telegram.org/bots/api#unpinallgeneralforumtopicmessages

Returns
instance of method *aiogram.methods.unpin_all_general_forum_topic_messages.UnpinAllGeneralForumTopicMessages*

ChatAdministratorRights

class aiogram.types.chat_administrator_rights.**ChatAdministratorRights**(**, is_anonymous: bool, can_manage_chat: bool, can_delete_messages: bool, can_manage_video_chats: bool, can_restrict_members: bool, can_promote_members: bool, can_change_info: bool, can_invite_users: bool, can_post_stories: bool, can_edit_stories: bool, can_delete_stories: bool, can_post_messages: bool | None = None, can_edit_messages: bool | None = None, can_pin_messages: bool | None = None, can_manage_topics: bool | None = None, **extra_data: Any*)

Represents the rights of an administrator in a chat.

Source: https://core.telegram.org/bots/api#chatadministratorrights

is_anonymous: bool

`True`, if the user's presence in the chat is hidden

can_manage_chat: bool

`True`, if the administrator can access the chat event log, get boost list, see hidden supergroup and channel members, report spam messages and ignore slow mode. Implied by any other administrator privilege.

can_delete_messages: bool

`True`, if the administrator can delete messages of other users

can_manage_video_chats: bool

`True`, if the administrator can manage video chats

can_restrict_members: bool

`True`, if the administrator can restrict, ban or unban chat members, or access supergroup statistics

can_promote_members: bool

`True`, if the administrator can add new administrators with a subset of their own privileges or demote administrators that they have promoted, directly or indirectly (promoted by administrators that were appointed by the user)

can_change_info: bool

`True`, if the user is allowed to change the chat title, photo and other settings

can_invite_users: bool

`True`, if the user is allowed to invite new users to the chat

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

can_post_stories: bool

True, if the administrator can post stories to the chat

can_edit_stories: bool

True, if the administrator can edit stories posted by other users, post stories to the chat page, pin chat stories, and access the chat's story archive

can_delete_stories: bool

True, if the administrator can delete stories posted by other users

can_post_messages: bool | None

Optional. True, if the administrator can post messages in the channel, or access channel statistics; for channels only

can_edit_messages: bool | None

Optional. True, if the administrator can edit messages of other users and can pin messages; for channels only

can_pin_messages: bool | None

Optional. True, if the user is allowed to pin messages; for groups and supergroups only

can_manage_topics: bool | None

Optional. True, if the user is allowed to create, rename, close, and reopen forum topics; for supergroups only

ChatBackground

class aiogram.types.chat_background.**ChatBackground**(**, type: BackgroundTypeFill | BackgroundTypeWallpaper | BackgroundTypePattern | BackgroundTypeChatTheme, **extra_data: Any*)

This object represents a chat background.

Source: https://core.telegram.org/bots/api#chatbackground

type: ***BackgroundTypeFill* | *BackgroundTypeWallpaper* | *BackgroundTypePattern* | *BackgroundTypeChatTheme***

Type of the background

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ChatBoost

class aiogram.types.chat_boost.**ChatBoost**(*, *boost_id: str*, *add_date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *expiration_date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *source:* ChatBoostSourcePremium | ChatBoostSourceGiftCode | ChatBoostSourceGiveaway, ***extra_data: Any*)

This object contains information about a chat boost.

Source: https://core.telegram.org/bots/api#chatboost

boost_id: str

Unique identifier of the boost

add_date: DateTime

Point in time (Unix timestamp) when the chat was boosted

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

expiration_date: DateTime

Point in time (Unix timestamp) when the boost will automatically expire, unless the booster's Telegram Premium subscription is prolonged

source: *ChatBoostSourcePremium* **|** *ChatBoostSourceGiftCode* **|** *ChatBoostSourceGiveaway*

Source of the added boost

ChatBoostAdded

class aiogram.types.chat_boost_added.**ChatBoostAdded**(*, *boost_count: int*, ***extra_data: Any*)

This object represents a service message about a user boosting a chat.

Source: https://core.telegram.org/bots/api#chatboostadded

boost_count: int

Number of boosts added by the user

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ChatBoostRemoved

class aiogram.types.chat_boost_removed.**ChatBoostRemoved**(**, chat: Chat, boost_id: str, remove_date: _datetime_serializer, return_type=int, when_used=unless - none)], source: ChatBoostSourcePremium | ChatBoostSourceGiftCode | ChatBoostSourceGiveaway, **extra_data: Any*)

This object represents a boost removed from a chat.

Source: https://core.telegram.org/bots/api#chatboostremoved

chat: *Chat*

Chat which was boosted

boost_id: str

Unique identifier of the boost

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

remove_date: DateTime

Point in time (Unix timestamp) when the boost was removed

source: *ChatBoostSourcePremium* **|** *ChatBoostSourceGiftCode* **|** *ChatBoostSourceGiveaway*

Source of the removed boost

ChatBoostSource

class aiogram.types.chat_boost_source.**ChatBoostSource**(***extra_data: Any*)

This object describes the source of a chat boost. It can be one of

- *aiogram.types.chat_boost_source_premium.ChatBoostSourcePremium*
- *aiogram.types.chat_boost_source_gift_code.ChatBoostSourceGiftCode*
- *aiogram.types.chat_boost_source_giveaway.ChatBoostSourceGiveaway*

Source: https://core.telegram.org/bots/api#chatboostsource

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ChatBoostSourceGiftCode

`class aiogram.types.chat_boost_source_gift_code.`**`ChatBoostSourceGiftCode`**`(`*, *source: Literal[ChatBoostSourceType.GIFT_CODE = ChatBoostSourceType.GIFT_CODE*, *user:* User, ***extra_data: Any*`)`

The boost was obtained by the creation of Telegram Premium gift codes to boost a chat. Each such code boosts the chat 4 times for the duration of the corresponding Telegram Premium subscription.

Source: https://core.telegram.org/bots/api#chatboostsourcegiftcode

`source: Literal[ChatBoostSourceType.GIFT_CODE]`

Source of the boost, always 'gift_code'

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init``(`*context: Any*, */*`)` → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`user:` *`User`*

User for which the gift code was created

ChatBoostSourceGiveaway

`class aiogram.types.chat_boost_source_giveaway.`**`ChatBoostSourceGiveaway`**`(`*, *source: Literal[ChatBoostSourceType.GIVEAWAY] = ChatBoostSourceType.GIVEAWAY*, *giveaway_message_id: int*, *user:* User | *None = None*, *prize_star_count: int | None = None*, *is_unclaimed: bool | None = None*, ***extra_data: Any*`)`

The boost was obtained by the creation of a Telegram Premium or a Telegram Star giveaway. This boosts the chat 4 times for the duration of the corresponding Telegram Premium subscription for Telegram Premium giveaways and *prize_star_count* / 500 times for one year for Telegram Star giveaways.

Source: https://core.telegram.org/bots/api#chatboostsourcegiveaway

`source: Literal[ChatBoostSourceType.GIVEAWAY]`

Source of the boost, always 'giveaway'

`giveaway_message_id: int`

Identifier of a message in the chat with the giveaway; the message could have been deleted already. May be 0 if the message isn't sent yet.

`user:` *`User`* **`| None`**

Optional. User that won the prize in the giveaway if any; for Telegram Premium giveaways only

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

prize_star_count: int | None

Optional. The number of Telegram Stars to be split between giveaway winners; for Telegram Star giveaways only

is_unclaimed: bool | None

Optional. True, if the giveaway was completed, but there was no user to win the prize

ChatBoostSourcePremium

class aiogram.types.chat_boost_source_premium.**ChatBoostSourcePremium**(**, source: Literal[ChatBoostSourceType.PREMIUM] = ChatBoostSourceType.PREMIUM, user: User, **extra_data: Any*)

The boost was obtained by subscribing to Telegram Premium or by gifting a Telegram Premium subscription to another user.

Source: https://core.telegram.org/bots/api#chatboostsourcepremium

source: Literal[ChatBoostSourceType.PREMIUM]

Source of the boost, always 'premium'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user: *User*

User that boosted the chat

ChatBoostUpdated

class aiogram.types.chat_boost_updated.**ChatBoostUpdated**(**, chat: Chat, boost: ChatBoost, **extra_data: Any*)

This object represents a boost added to a chat or changed.

Source: https://core.telegram.org/bots/api#chatboostupdated

chat: *Chat*

Chat which was boosted

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

boost: *ChatBoost*

Information about the chat boost

ChatFullInfo

class aiogram.types.chat_full_info.**ChatFullInfo**(**, id: int, type: str, title: str | None = None, username: str | None = None, first_name: str | None = None, last_name: str | None = None, is_forum: bool | None = None, accent_color_id: int, active_usernames: List[str] | None = None, available_reactions: List[ReactionTypeEmoji | ReactionTypeCustomEmoji | ReactionTypePaid] | None = None, background_custom_emoji_id: str | None = None, bio: str | None = None, birthdate: Birthdate | None = None, business_intro: BusinessIntro | None = None, business_location: BusinessLocation | None = None, business_opening_hours: BusinessOpeningHours | None = None, can_set_sticker_set: bool | None = None, custom_emoji_sticker_set_name: str | None = None, description: str | None = None, emoji_status_custom_emoji_id: str | None = None, emoji_status_expiration_date: _datetime_serializer, return_type=int, when_used=unless - none)] | None = None, has_aggressive_anti_spam_enabled: bool | None = None, has_hidden_members: bool | None = None, has_private_forwards: bool | None = None, has_protected_content: bool | None = None, has_restricted_voice_and_video_messages: bool | None = None, has_visible_history: bool | None = None, invite_link: str | None = None, join_by_request: bool | None = None, join_to_send_messages: bool | None = None, linked_chat_id: int | None = None, location: ChatLocation | None = None, message_auto_delete_time: int | None = None, permissions: ChatPermissions | None = None, personal_chat: Chat | None = None, photo: ChatPhoto | None = None, pinned_message: Message | None = None, profile_accent_color_id: int | None = None, profile_background_custom_emoji_id: str | None = None, slow_mode_delay: int | None = None, sticker_set_name: str | None = None, unrestrict_boost_count: int | None = None, max_reaction_count: int, can_send_paid_media: bool | None = None, **extra_data: Any*)

This object contains full information about a chat.

Source: https://core.telegram.org/bots/api#chatfullinfo

id: int

Unique identifier for this chat. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this identifier.

type: str

Type of the chat, can be either 'private', 'group', 'supergroup' or 'channel'

accent_color_id: int

Identifier of the accent color for the chat name and backgrounds of the chat photo, reply header, and link preview. See accent colors for more details.

max_reaction_count: int

The maximum number of reactions that can be set on a message in the chat

title: str | None

Optional. Title, for supergroups, channels and group chats

username: str | None

Optional. Username, for private chats, supergroups and channels if available

first_name: str | None

Optional. First name of the other party in a private chat

last_name: str | None

Optional. Last name of the other party in a private chat

is_forum: bool | None

Optional. `True`, if the supergroup chat is a forum (has topics enabled)

photo: *ChatPhoto* | None

Optional. Chat photo

active_usernames: List[str] | None

Optional. If non-empty, the list of all active chat usernames; for private chats, supergroups and channels

birthdate: *Birthdate* | None

Optional. For private chats, the date of birth of the user

business_intro: *BusinessIntro* | None

Optional. For private chats with business accounts, the intro of the business

business_location: *BusinessLocation* | None

Optional. For private chats with business accounts, the location of the business

business_opening_hours: *BusinessOpeningHours* | None

Optional. For private chats with business accounts, the opening hours of the business

personal_chat: *Chat* | None

Optional. For private chats, the personal channel of the user

available_reactions: List[*ReactionTypeEmoji* | *ReactionTypeCustomEmoji* | *ReactionTypePaid*] | None

Optional. List of available reactions allowed in the chat. If omitted, then all emoji reactions are allowed.

background_custom_emoji_id: str | None

Optional. Custom emoji identifier of the emoji chosen by the chat for the reply header and link preview background

profile_accent_color_id: int | None

Optional. Identifier of the accent color for the chat's profile background. See profile accent colors for more details.

profile_background_custom_emoji_id: str | None

Optional. Custom emoji identifier of the emoji chosen by the chat for its profile background

emoji_status_custom_emoji_id: str | None

Optional. Custom emoji identifier of the emoji status of the chat or the other party in a private chat

emoji_status_expiration_date: DateTime | None

Optional. Expiration date of the emoji status of the chat or the other party in a private chat, in Unix time, if any

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

bio: str | None

Optional. Bio of the other party in a private chat

has_private_forwards: bool | None

Optional. `True`, if privacy settings of the other party in the private chat allows to use `tg://user?id=<user_id>` links only in chats with the user

has_restricted_voice_and_video_messages: bool | None

Optional. `True`, if the privacy settings of the other party restrict sending voice and video note messages in the private chat

join_to_send_messages: bool | None

Optional. `True`, if users need to join the supergroup before they can send messages

join_by_request: bool | None

Optional. `True`, if all users directly joining the supergroup without using an invite link need to be approved by supergroup administrators

description: str | None

Optional. Description, for groups, supergroups and channel chats

invite_link: str | None

Optional. Primary invite link, for groups, supergroups and channel chats

pinned_message: *Message* | None

Optional. The most recent pinned message (by sending date)

permissions: *ChatPermissions* | None

Optional. Default chat member permissions, for groups and supergroups

can_send_paid_media: bool | None

Optional. `True`, if paid media messages can be sent or forwarded to the channel chat. The field is available only for channel chats.

slow_mode_delay: int | None

Optional. For supergroups, the minimum allowed delay between consecutive messages sent by each unprivileged user; in seconds

unrestrict_boost_count: int | None

Optional. For supergroups, the minimum number of boosts that a non-administrator user needs to add in order to ignore slow mode and chat permissions

message_auto_delete_time: int | None

Optional. The time after which all messages sent to the chat will be automatically deleted; in seconds

has_aggressive_anti_spam_enabled: bool | None

Optional. `True`, if aggressive anti-spam checks are enabled in the supergroup. The field is only available to chat administrators.

has_hidden_members: bool | None

Optional. `True`, if non-administrators can only get the list of bots and administrators in the chat

has_protected_content: bool | None

Optional. `True`, if messages from the chat can't be forwarded to other chats

has_visible_history: bool | None

Optional. `True`, if new chat members will have access to old messages; available only to chat administrators

sticker_set_name: str | None

Optional. For supergroups, name of the group sticker set

can_set_sticker_set: bool | None

Optional. `True`, if the bot can change the group sticker set

custom_emoji_sticker_set_name: str | None

Optional. For supergroups, the name of the group's custom emoji sticker set. Custom emoji from this set can be used by all users and bots in the group.

linked_chat_id: int | None

Optional. Unique identifier for the linked chat, i.e. the discussion group identifier for a channel and vice versa; for supergroups and channel chats. This identifier may be greater than 32 bits and some programming languages may have difficulty/silent defects in interpreting it. But it is smaller than 52 bits, so a signed 64 bit integer or double-precision float type are safe for storing this identifier.

location: *ChatLocation* | None

Optional. For supergroups, the location to which the supergroup is connected

ChatInviteLink

class aiogram.types.chat_invite_link.**ChatInviteLink**(**, invite_link: str, creator: User, creates_join_request: bool, is_primary: bool, is_revoked: bool, name: str | None = None, expire_date: _datetime_serializer, return_type=int, when_used=unless - none)] | None = None, member_limit: int | None = None, pending_join_request_count: int | None = None, subscription_period: int | None = None, subscription_price: int | None = None, **extra_data: Any*)

Represents an invite link for a chat.

Source: https://core.telegram.org/bots/api#chatinvitelink

invite_link: str

The invite link. If the link was created by another chat administrator, then the second part of the link will be replaced with '…'.

creator: *User*

Creator of the link

creates_join_request: bool

True, if users joining the chat via the link need to be approved by chat administrators

is_primary: bool

True, if the link is primary

is_revoked: bool

True, if the link is revoked

name: str | None

Optional. Invite link name

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

expire_date: DateTime | None

Optional. Point in time (Unix timestamp) when the link will expire or has been expired

member_limit: int | None

Optional. The maximum number of users that can be members of the chat simultaneously after joining the chat via this invite link; 1-99999

pending_join_request_count: int | None

Optional. Number of pending join requests created using this link

subscription_period: int | None

Optional. The number of seconds the subscription will be active for before the next payment

subscription_price: int | None

Optional. The amount of Telegram Stars a user must pay initially and after each subsequent subscription period to be a member of the chat using the link

ChatJoinRequest

class aiogram.types.chat_join_request.**ChatJoinRequest**(**, chat: Chat, from_user: User, user_chat_id: int, date: _datetime_serializer, return_type=int, when_used=unless - none)], bio: str | None = None, invite_link: ChatInviteLink | None = None, **extra_data: Any*)

Represents a join request sent to a chat.

Source: https://core.telegram.org/bots/api#chatjoinrequest

chat: *Chat*

Chat to which the request was sent

from_user: *User*

User that sent the join request

user_chat_id: int

Identifier of a private chat with the user who sent the join request. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a 64-bit integer or double-precision float type are safe for storing this identifier. The bot can use this identifier for 5 minutes to send messages until the join request is processed, assuming no other administrator contacted the user.

date: DateTime

Date the request was sent in Unix time

bio: str | None

Optional. Bio of the user.

invite_link: ChatInviteLink | None

Optional. Chat invite link that was used by the user to send the join request

approve(***kwargs: Any*) → *ApproveChatJoinRequest*

Shortcut for method `aiogram.methods.approve_chat_join_request.ApproveChatJoinRequest` will automatically fill method attributes:

- `chat_id`
- `user_id`

Use this method to approve a chat join request. The bot must be an administrator in the chat for this to work and must have the *can_invite_users* administrator right. Returns `True` on success.

Source: https://core.telegram.org/bots/api#approvechatjoinrequest

Returns
instance of method `aiogram.methods.approve_chat_join_request.ApproveChatJoinRequest`

decline(***kwargs: Any*) → *DeclineChatJoinRequest*

Shortcut for method `aiogram.methods.decline_chat_join_request.DeclineChatJoinRequest` will automatically fill method attributes:

- `chat_id`
- `user_id`

Use this method to decline a chat join request. The bot must be an administrator in the chat for this to work and must have the *can_invite_users* administrator right. Returns `True` on success.

Source: https://core.telegram.org/bots/api#declinechatjoinrequest

Returns
instance of method `aiogram.methods.decline_chat_join_request.DeclineChatJoinRequest`

answer(*text: str, business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, entities: Optional[List[MessageEntity]] = None, link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMessage*

Shortcut for method `aiogram.methods.send_message.SendMessage` will automatically fill method attributes:

- `chat_id`

Use this method to send text messages. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendmessage

Parameters

- **`text`** – Text of the message to be sent, 1-4096 characters after entities parsing
- **`business_connection_id`** – Unique identifier of the business connection on behalf of which the message will be sent
- **`message_thread_id`** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **`parse_mode`** – Mode for parsing entities in the message text. See formatting options for more details.
- **`entities`** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*
- **`link_preview_options`** – Link preview generation options for the message
- **`disable_notification`** – Sends the message silently. Users will receive a notification with no sound.
- **`protect_content`** – Protects the contents of the sent message from forwarding and saving
- **`message_effect_id`** – Unique identifier of the message effect to be added to the message; for private chats only
- **`reply_parameters`** – Description of the message to reply to
- **`reply_markup`** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **`allow_sending_without_reply`** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **`disable_web_page_preview`** – Disables link previews for links in this message
- **`reply_to_message_id`** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_message.SendMessage`

`answer_pm`(*text: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>*, *entities: Optional[List[MessageEntity]] = None*, *link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → SendMessage

Shortcut for method *aiogram.methods.send_message.SendMessage* will automatically fill method attributes:

- chat_id

Use this method to send text messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendmessage

Parameters

- **text** – Text of the message to be sent, 1-4096 characters after entities parsing
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **parse_mode** – Mode for parsing entities in the message text. See formatting options for more details.
- **entities** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*
- **link_preview_options** – Link preview generation options for the message
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass True if the message should be sent even if the specified replied-to message is not found
- **disable_web_page_preview** – Disables link previews for links in this message
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_message.SendMessage*

answer_animation(*animation: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAnimation*

Shortcut for method `aiogram.methods.send_animation.SendAnimation` will automatically fill method attributes:

- `chat_id`

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

Parameters

- **animation** – Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent animation in seconds
- **width** – Animation width
- **height** – Animation height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the animation caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the animation needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_animation.SendAnimation*

answer_animation_pm(*animation: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAnimation*

Shortcut for method *aiogram.methods.send_animation.SendAnimation* will automatically fill method attributes:

- `chat_id`

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

Parameters

- **animation** – Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent

- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent animation in seconds
- **width** – Animation width
- **height** – Animation height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the animation caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the animation needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_animation.SendAnimation`

answer_audio(*audio: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, performer: Optional[str] = None, title: Optional[str] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAudio*

Shortcut for method *aiogram.methods.send_audio.SendAudio* will automatically fill method attributes:

- chat_id

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the *aiogram.methods.send_voice.SendVoice* method instead.

Source: https://core.telegram.org/bots/api#sendaudio

Parameters

- **audio** – Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Audio caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the audio caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_audio.SendAudio*

answer_audio_pm(*audio: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, performer: Optional[str] = None, title: Optional[str] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAudio*

Shortcut for method *aiogram.methods.send_audio.SendAudio* will automatically fill method attributes:

- chat_id

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the *aiogram.methods.send_voice.SendVoice* method instead.

Source: https://core.telegram.org/bots/api#sendaudio

Parameters

- **audio** – Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Audio caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the audio caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded

using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_audio.SendAudio`

answer_contact(*phone_number: str*, *first_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendContact*

Shortcut for method `aiogram.methods.send_contact.SendContact` will automatically fill method attributes:

- `chat_id`

Use this method to send phone contacts. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendcontact

Parameters

- **phone_number** – Contact's phone number
- **first_name** – Contact's first name
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **last_name** – Contact's last name
- **vcard** – Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving

- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_contact.SendContact*

answer_contact_pm(*phone_number: str*, *first_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendContact*

Shortcut for method *aiogram.methods.send_contact.SendContact* will automatically fill method attributes:

- `chat_id`

Use this method to send phone contacts. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendcontact

Parameters

- **phone_number** – Contact's phone number
- **first_name** – Contact's first name
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **last_name** – Contact's last name
- **vcard** – Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to

- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_contact.SendContact*

answer_document(*document: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, disable_content_type_detection: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendDocument*

Shortcut for method *aiogram.methods.send_document.SendDocument* will automatically fill method attributes:

- `chat_id`

Use this method to send general files. On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

Parameters

- **document** – File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the document caption. See formatting options for more details.

- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_document.SendDocument`

answer_document_pm(*document: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, disable_content_type_detection: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendDocument*

Shortcut for method `aiogram.methods.send_document.SendDocument` will automatically fill method attributes:

- `chat_id`

Use this method to send general files. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

Parameters

- **document** – File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent

- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the document caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_document.SendDocument`

answer_game(*game_short_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendGame*

Shortcut for method `aiogram.methods.send_game.SendGame` will automatically fill method attributes:

- `chat_id`

Use this method to send a game. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendgame

Parameters

- **game_short_name** – Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_game.SendGame`

answer_game_pm(*game_short_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendGame*

Shortcut for method `aiogram.methods.send_game.SendGame` will automatically fill method attributes:

- `chat_id`

Use this method to send a game. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendgame

Parameters

- **game_short_name** – Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_game.SendGame`

answer_invoice(*title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], message_thread_id: Optional[int] = None, provider_token: Optional[str] = None, max_tip_amount: Optional[int] = None, suggested_tip_amounts: Optional[List[int]] = None, start_parameter: Optional[str] = None, provider_data: Optional[str] = None, photo_url: Optional[str] = None, photo_size: Optional[int] = None, photo_width: Optional[int] = None, photo_height: Optional[int] = None, need_name: Optional[bool] = None, need_phone_number: Optional[bool] = None, need_email: Optional[bool] = None, need_shipping_address: Optional[bool] = None, send_phone_number_to_provider: Optional[bool] = None, send_email_to_provider: Optional[bool] = None, is_flexible: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendInvoice*

Shortcut for method `aiogram.methods.send_invoice.SendInvoice` will automatically fill method attributes:

- `chat_id`

Use this method to send invoices. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **currency** – Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.
- **prices** – Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **provider_token** – Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.
- **max_tip_amount** – The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts

can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.

- **start_parameter** – Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter
- **provider_data** – JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.
- **need_phone_number** – Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.
- **need_email** – Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.
- **need_shipping_address** – Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.
- **send_phone_number_to_provider** – Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.
- **send_email_to_provider** – Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.
- **is_flexible** – Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_invoice.SendInvoice*

answer_invoice_pm(*title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], message_thread_id: Optional[int] = None, provider_token: Optional[str] = None, max_tip_amount: Optional[int] = None, suggested_tip_amounts: Optional[List[int]] = None, start_parameter: Optional[str] = None, provider_data: Optional[str] = None, photo_url: Optional[str] = None, photo_size: Optional[int] = None, photo_width: Optional[int] = None, photo_height: Optional[int] = None, need_name: Optional[bool] = None, need_phone_number: Optional[bool] = None, need_email: Optional[bool] = None, need_shipping_address: Optional[bool] = None, send_phone_number_to_provider: Optional[bool] = None, send_email_to_provider: Optional[bool] = None, is_flexible: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendInvoice*

Shortcut for method `aiogram.methods.send_invoice.SendInvoice` will automatically fill method attributes:

- `chat_id`

Use this method to send invoices. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **currency** – Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.
- **prices** – Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **provider_token** – Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.
- **max_tip_amount** – The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.
- **start_parameter** – Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent

message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter

- **provider_data** – JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.
- **need_phone_number** – Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.
- **need_email** – Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.
- **need_shipping_address** – Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.
- **send_phone_number_to_provider** – Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.
- **send_email_to_provider** – Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.
- **is_flexible** – Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_invoice.SendInvoice*

answer_location(*latitude: float, longitude: float, business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, horizontal_accuracy: Optional[float] = None, live_period: Optional[int] = None, heading: Optional[int] = None, proximity_alert_radius: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendLocation*

Shortcut for method `aiogram.methods.send_location.SendLocation` will automatically fill method attributes:

- `chat_id`

Use this method to send point on the map. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendlocation

Parameters

- **latitude** – Latitude of the location
- **longitude** – Longitude of the location
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **live_period** – Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.
- **heading** – For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_location.SendLocation*

answer_location_pm(*latitude: float*, *longitude: float*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *live_period: Optional[int] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendLocation*

Shortcut for method *aiogram.methods.send_location.SendLocation* will automatically fill method attributes:

- chat_id

Use this method to send point on the map. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendlocation

Parameters

- **latitude** – Latitude of the location
- **longitude** – Longitude of the location
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **live_period** – Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.
- **heading** – For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to

- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_location.SendLocation`

answer_media_group(*media: List[Union[InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMediaGroup*

Shortcut for method `aiogram.methods.send_media_group.SendMediaGroup` will automatically fill method attributes:

- `chat_id`

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **media** – A JSON-serialized array describing messages to be sent, must include 2-10 items
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends messages silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent messages from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the messages are a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_media_group.SendMediaGroup`

answer_media_group_pm(*media: List[Union[InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMediaGroup*

Shortcut for method `aiogram.methods.send_media_group.SendMediaGroup` will automatically fill method attributes:

- `chat_id`

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **media** – A JSON-serialized array describing messages to be sent, must include 2-10 items
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends messages silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent messages from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the messages are a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_media_group.SendMediaGroup`

answer_photo(*photo: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPhoto*

Shortcut for method `aiogram.methods.send_photo.SendPhoto` will automatically fill method attributes:

- `chat_id`

Use this method to send photos. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendphoto

Parameters

- **photo** – Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the photo caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the photo needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_photo.SendPhoto*

answer_photo_pm(*photo: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPhoto*

Shortcut for method `aiogram.methods.send_photo.SendPhoto` will automatically fill method attributes:

- `chat_id`

Use this method to send photos. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendphoto

Parameters

- **photo** – Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the photo caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the photo needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_photo.SendPhoto*

answer_poll(*question: str, options: List[Union[InputPollOption, str]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, question_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, question_entities: Optional[List[MessageEntity]] = None, is_anonymous: Optional[bool] = None, type: Optional[str] = None, allows_multiple_answers: Optional[bool] = None, correct_option_id: Optional[int] = None, explanation: Optional[str] = None, explanation_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, explanation_entities: Optional[List[MessageEntity]] = None, open_period: Optional[int] = None, close_date: Optional[Union[datetime.datetime, datetime.timedelta, int]] = None, is_closed: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPoll*

Shortcut for method *aiogram.methods.send_poll.SendPoll* will automatically fill method attributes:

- `chat_id`

Use this method to send a native poll. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendpoll

Parameters

- **question** – Poll question, 1-300 characters
- **options** – A JSON-serialized list of 2-10 answer options
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **question_parse_mode** – Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed
- **question_entities** – A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*
- **is_anonymous** – `True`, if the poll needs to be anonymous, defaults to `True`
- **type** – Poll type, 'quiz' or 'regular', defaults to 'regular'
- **allows_multiple_answers** – `True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`
- **correct_option_id** – 0-based identifier of the correct answer option, required for polls in quiz mode
- **explanation** – Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing

- **explanation_parse_mode** – Mode for parsing entities in the explanation. See formatting options for more details.
- **explanation_entities** – A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*
- **open_period** – Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.
- **close_date** – Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.
- **is_closed** – Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_poll.SendPoll`

answer_poll_pm(*question: str, options: List[Union[InputPollOption, str]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, question_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, question_entities: Optional[List[MessageEntity]] = None, is_anonymous: Optional[bool] = None, type: Optional[str] = None, allows_multiple_answers: Optional[bool] = None, correct_option_id: Optional[int] = None, explanation: Optional[str] = None, explanation_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, explanation_entities: Optional[List[MessageEntity]] = None, open_period: Optional[int] = None, close_date: Optional[Union[datetime.datetime, datetime.timedelta, int]] = None, is_closed: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPoll*

Shortcut for method `aiogram.methods.send_poll.SendPoll` will automatically fill method attributes:

- `chat_id`

Use this method to send a native poll. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpoll

Parameters

- **question** – Poll question, 1-300 characters
- **options** – A JSON-serialized list of 2-10 answer options
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **question_parse_mode** – Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed
- **question_entities** – A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*
- **is_anonymous** – `True`, if the poll needs to be anonymous, defaults to `True`
- **type** – Poll type, 'quiz' or 'regular', defaults to 'regular'
- **allows_multiple_answers** – `True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`
- **correct_option_id** – 0-based identifier of the correct answer option, required for polls in quiz mode
- **explanation** – Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing
- **explanation_parse_mode** – Mode for parsing entities in the explanation. See formatting options for more details.
- **explanation_entities** – A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*
- **open_period** – Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.
- **close_date** – Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.
- **is_closed** – Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_poll.SendPoll`

answer_dice(*business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *emoji: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendDice*

Shortcut for method `aiogram.methods.send_dice.SendDice` will automatically fill method attributes:

- `chat_id`

Use this method to send an animated emoji that will display a random value. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#senddice

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_dice.SendDice`

answer_dice_pm(*business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *emoji: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendDice*

Shortcut for method `aiogram.methods.send_dice.SendDice` will automatically fill method attributes:

- `chat_id`

Use this method to send an animated emoji that will display a random value. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#senddice

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_dice.SendDice`

answer_sticker(*sticker: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendSticker*

Shortcut for method `aiogram.methods.send_sticker.SendSticker` will automatically fill method attributes:

- `chat_id`

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendsticker

Parameters

- **sticker** – Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji associated with the sticker; only for just uploaded stickers
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_sticker.SendSticker`

answer_sticker_pm(*sticker: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendSticker*

Shortcut for method `aiogram.methods.send_sticker.SendSticker` will automatically fill method attributes:

- `chat_id`

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendsticker

Parameters

- **sticker** – Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji associated with the sticker; only for just uploaded stickers
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_sticker.SendSticker`

answer_venue(*latitude: float*, *longitude: float*, *title: str*, *address: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendVenue*

Shortcut for method `aiogram.methods.send_venue.SendVenue` will automatically fill method attributes:

- `chat_id`

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

Parameters

- **latitude** – Latitude of the venue
- **longitude** – Longitude of the venue
- **title** – Name of the venue
- **address** – Address of the venue
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

- **foursquare_id** – Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)
- **google_place_id** – Google Places identifier of the venue
- **google_place_type** – Google Places type of the venue. (See supported types.)
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_venue.SendVenue`

answer_venue_pm(*latitude: float*, *longitude: float*, *title: str*, *address: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendVenue*

Shortcut for method `aiogram.methods.send_venue.SendVenue` will automatically fill method attributes:

- `chat_id`

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

Parameters

- **latitude** – Latitude of the venue
- **longitude** – Longitude of the venue
- **title** – Name of the venue
- **address** – Address of the venue
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent

- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **foursquare_id** – Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)
- **google_place_id** – Google Places identifier of the venue
- **google_place_type** – Google Places type of the venue. (See supported types.)
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_venue.SendVenue`

answer_video(*video: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, supports_streaming: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideo*

Shortcut for method `aiogram.methods.send_video.SendVideo` will automatically fill method attributes:

- `chat_id`

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

Parameters

- **video** – Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the

Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the video needs to be covered with a spoiler animation
- **supports_streaming** – Pass `True` if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_video.SendVideo`

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

answer_video_pm(*video: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, supports_streaming: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideo*

Shortcut for method `aiogram.methods.send_video.SendVideo` will automatically fill method attributes:

- `chat_id`

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

Parameters

- **video** – Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing

- **parse_mode** – Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the video needs to be covered with a spoiler animation
- **supports_streaming** – Pass `True` if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_video.SendVideo*

answer_video_note(*video_note: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, length: Optional[int] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideoNote*

Shortcut for method *aiogram.methods.send_video_note.SendVideoNote* will automatically fill method attributes:

- `chat_id`

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **video_note** – Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **length** – Video width and height, i.e. diameter of the video message
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_video_note.SendVideoNote*

answer_video_note_pm(*video_note: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, length: Optional[int] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideoNote*

Shortcut for method *aiogram.methods.send_video_note.SendVideoNote* will automatically fill method attributes:

- `chat_id`

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **video_note** – Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **length** – Video width and height, i.e. diameter of the video message
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_video_note.SendVideoNote`

answer_voice(*voice: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVoice*

Shortcut for method `aiogram.methods.send_voice.SendVoice` will automatically fill method attributes:

- `chat_id`

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as `aiogram.types.audio.Audio` or `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

Parameters

- **voice** – Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Voice message caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the voice message caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the voice message in seconds
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_voice.SendVoice`

answer_voice_pm(*voice: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVoice*

Shortcut for method `aiogram.methods.send_voice.SendVoice` will automatically fill method attributes:

- chat_id

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as *aiogram.types.audio.Audio* or *aiogram.types.document.Document*). On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

Parameters

- **voice** – Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Voice message caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the voice message caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the voice message in seconds
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass True if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_voice.SendVoice*

ChatLocation

class aiogram.types.chat_location.**ChatLocation**(**, location: Location, address: str, **extra_data: Any*)

Represents a location to which a chat is connected.

Source: https://core.telegram.org/bots/api#chatlocation

location: *Location*

The location to which the supergroup is connected. Can't be a live location.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

address: str

Location address; 1-64 characters, as defined by the chat owner

ChatMember

class aiogram.types.chat_member.**ChatMember**(***extra_data: Any*)

This object contains information about one member of a chat. Currently, the following 6 types of chat members are supported:

- *aiogram.types.chat_member_owner.ChatMemberOwner*
- *aiogram.types.chat_member_administrator.ChatMemberAdministrator*
- *aiogram.types.chat_member_member.ChatMemberMember*
- *aiogram.types.chat_member_restricted.ChatMemberRestricted*
- *aiogram.types.chat_member_left.ChatMemberLeft*
- *aiogram.types.chat_member_banned.ChatMemberBanned*

Source: https://core.telegram.org/bots/api#chatmember

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ChatMemberAdministrator

class aiogram.types.chat_member_administrator.**ChatMemberAdministrator**(**, status: Literal[ChatMemberStatus.ADMINISTRATOR = ChatMemberStatus.ADMINISTRATOR, user: User, can_be_edited: bool, is_anonymous: bool, can_manage_chat: bool, can_delete_messages: bool, can_manage_video_chats: bool, can_restrict_members: bool, can_promote_members: bool, can_change_info: bool, can_invite_users: bool, can_post_stories: bool, can_edit_stories: bool, can_delete_stories: bool, can_post_messages: bool | None = None, can_edit_messages: bool | None = None, can_pin_messages: bool | None = None, can_manage_topics: bool | None = None, custom_title: str | None = None, **extra_data: Any*)

Represents a chat member that has some additional privileges.

Source: https://core.telegram.org/bots/api#chatmemberadministrator

status: Literal[ChatMemberStatus.ADMINISTRATOR]

The member's status in the chat, always 'administrator'

user: *User*

Information about the user

can_be_edited: bool

`True`, if the bot is allowed to edit administrator privileges of that user

is_anonymous: bool

`True`, if the user's presence in the chat is hidden

can_manage_chat: bool

`True`, if the administrator can access the chat event log, get boost list, see hidden supergroup and channel members, report spam messages and ignore slow mode. Implied by any other administrator privilege.

can_delete_messages: bool

`True`, if the administrator can delete messages of other users `can_manage_video_chats: bool`

`True`, if the administrator can manage video chats

`can_restrict_members: bool`

`True`, if the administrator can restrict, ban or unban chat members, or access supergroup statistics

`can_promote_members: bool`

`True`, if the administrator can add new administrators with a subset of their own privileges or demote administrators that they have promoted, directly or indirectly (promoted by administrators that were appointed by the user)

`can_change_info: bool`

`True`, if the user is allowed to change the chat title, photo and other settings

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`can_invite_users: bool`

`True`, if the user is allowed to invite new users to the chat

`can_post_stories: bool`

`True`, if the administrator can post stories to the chat

`can_edit_stories: bool`

`True`, if the administrator can edit stories posted by other users, post stories to the chat page, pin chat stories, and access the chat's story archive

`can_delete_stories: bool`

`True`, if the administrator can delete stories posted by other users

`can_post_messages: bool | None`

Optional. `True`, if the administrator can post messages in the channel, or access channel statistics; for channels only

`can_edit_messages: bool | None`

Optional. `True`, if the administrator can edit messages of other users and can pin messages; for channels only

`can_pin_messages: bool | None`

Optional. `True`, if the user is allowed to pin messages; for groups and supergroups only

`can_manage_topics: bool | None`

Optional. `True`, if the user is allowed to create, rename, close, and reopen forum topics; for supergroups only

`custom_title: str | None`

Optional. Custom title for this user

ChatMemberBanned

class aiogram.types.chat_member_banned.**ChatMemberBanned**(**, status: Literal[ChatMemberStatus.KICKED] = ChatMemberStatus.KICKED, user: User, until_date: _datetime_serializer, return_type=int, when_used=unless - none)], **extra_data: Any*)

Represents a chat member that was banned in the chat and can't return to the chat or view chat messages.

Source: https://core.telegram.org/bots/api#chatmemberbanned

status: Literal[ChatMemberStatus.KICKED]

The member's status in the chat, always 'kicked'

user: *User*

Information about the user

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

until_date: DateTime

Date when restrictions will be lifted for this user; Unix time. If 0, then the user is banned forever

ChatMemberLeft

class aiogram.types.chat_member_left.**ChatMemberLeft**(**, status: Literal[ChatMemberStatus.LEFT] = ChatMemberStatus.LEFT, user: User, **extra_data: Any*)

Represents a chat member that isn't currently a member of the chat, but may join it themselves.

Source: https://core.telegram.org/bots/api#chatmemberleft

status: Literal[ChatMemberStatus.LEFT]

The member's status in the chat, always 'left'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user: *User*

Information about the user

ChatMemberMember

class aiogram.types.chat_member_member.**ChatMemberMember**(**, status: Literal[ChatMemberStatus.MEMBER] = ChatMemberStatus.MEMBER, user: User, until_date: _datetime_serializer, return_type=int, when_used=unless - none)] | None = None, **extra_data: Any*)

Represents a chat member that has no additional privileges or restrictions.

Source: https://core.telegram.org/bots/api#chatmembermember

status: Literal[ChatMemberStatus.MEMBER]

The member's status in the chat, always 'member'

user: *User*

Information about the user

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

until_date: DateTime | None

Optional. Date when the user's subscription will expire; Unix time

ChatMemberOwner

class aiogram.types.chat_member_owner.**ChatMemberOwner**(**, status: Literal[ChatMemberStatus.CREATOR] = ChatMemberStatus.CREATOR, user: User, is_anonymous: bool, custom_title: str | None = None, **extra_data: Any*)

Represents a chat member that owns the chat and has all administrator privileges.

Source: https://core.telegram.org/bots/api#chatmemberowner

status: Literal[ChatMemberStatus.CREATOR]

The member's status in the chat, always 'creator'

user: *User*

Information about the user

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

is_anonymous: bool

`True`, if the user's presence in the chat is hidden

custom_title: str | None

Optional. Custom title for this user

ChatMemberRestricted

class aiogram.types.chat_member_restricted.**ChatMemberRestricted**(**, status: Literal[ChatMemberStatus.RESTRICTED] = ChatMemberStatus.RESTRICTED, user: User, is_member: bool, can_send_messages: bool, can_send_audios: bool, can_send_documents: bool, can_send_photos: bool, can_send_videos: bool, can_send_video_notes: bool, can_send_voice_notes: bool, can_send_polls: bool, can_send_other_messages: bool, can_add_web_page_previews: bool, can_change_info: bool, can_invite_users: bool, can_pin_messages: bool, can_manage_topics: bool, until_date: _datetime_serializer, return_type=int, when_used=unless - none)], **extra_data: Any*)

Represents a chat member that is under certain restrictions in the chat. Supergroups only.

Source: https://core.telegram.org/bots/api#chatmemberrestricted

status: Literal[ChatMemberStatus.RESTRICTED]

The member's status in the chat, always 'restricted'

user: *User*

Information about the user

is_member: bool

True, if the user is a member of the chat at the moment of the request

can_send_messages: bool

True, if the user is allowed to send text messages, contacts, giveaways, giveaway winners, invoices, locations and venues

can_send_audios: bool

True, if the user is allowed to send audios

can_send_documents: bool

True, if the user is allowed to send documents

can_send_photos: bool

True, if the user is allowed to send photos

can_send_videos: bool

True, if the user is allowed to send videos

can_send_video_notes: bool

True, if the user is allowed to send video notes

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

`can_send_voice_notes: bool`
`True`, if the user is allowed to send voice notes

`can_send_polls: bool`
`True`, if the user is allowed to send polls

`can_send_other_messages: bool`
`True`, if the user is allowed to send animations, games, stickers and use inline bots

`can_add_web_page_previews: bool`
`True`, if the user is allowed to add web page previews to their messages

`can_change_info: bool`
`True`, if the user is allowed to change the chat title, photo and other settings

`can_invite_users: bool`
`True`, if the user is allowed to invite new users to the chat

`can_pin_messages: bool`
`True`, if the user is allowed to pin messages

`can_manage_topics: bool`
`True`, if the user is allowed to create forum topics

`until_date: DateTime`
Date when restrictions will be lifted for this user; Unix time. If 0, then the user is restricted forever

ChatMemberUpdated

`class aiogram.types.chat_member_updated.ChatMemberUpdated`(*, *chat:* Chat, *from_user:* User, *date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *old_chat_member:* ChatMemberOwner | ChatMemberAdministrator | ChatMemberMember | ChatMemberRestricted | ChatMemberLeft | ChatMemberBanned, *new_chat_member:* ChatMemberOwner | ChatMemberAdministrator | ChatMemberMember | ChatMemberRestricted | ChatMemberLeft | ChatMemberBanned, *invite_link:* ChatInviteLink | *None = None*, *via_join_request: bool* | *None = None*, *via_chat_folder_invite_link: bool* | *None = None*, ***extra_data: Any*)

This object represents changes in the status of a chat member.

Source: https://core.telegram.org/bots/api#chatmemberupdated

chat: *Chat*

Chat the user belongs to

from_user: *User*

Performer of the action, which resulted in the change

date: **DateTime**

Date the change was done in Unix time

old_chat_member: *ChatMemberOwner* | *ChatMemberAdministrator* | *ChatMemberMember* | *ChatMemberRestricted* | *ChatMemberLeft* | *ChatMemberBanned*

Previous information about the chat member

new_chat_member: *ChatMemberOwner* | *ChatMemberAdministrator* | *ChatMemberMember* | *ChatMemberRestricted* | *ChatMemberLeft* | *ChatMemberBanned*

New information about the chat member

invite_link: *ChatInviteLink* | **None**

Optional. Chat invite link, which was used by the user to join the chat; for joining by invite link events only.

via_join_request: **bool | None**

Optional. True, if the user joined the chat after sending a direct join request without using an invite link and being approved by an administrator

via_chat_folder_invite_link: **bool | None**

Optional. True, if the user joined the chat via a chat folder invite link

answer(*text: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>*, *entities: Optional[List[MessageEntity]] = None*, *link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendMessage*

Shortcut for method `aiogram.methods.send_message.SendMessage` will automatically fill method attributes:

- `chat_id`

Use this method to send text messages. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendmessage

Parameters

- **text** – Text of the message to be sent, 1-4096 characters after entities parsing
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

- **parse_mode** – Mode for parsing entities in the message text. See formatting options for more details.
- **entities** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*
- **link_preview_options** – Link preview generation options for the message
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **disable_web_page_preview** – Disables link previews for links in this message
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_message.SendMessage`

answer_animation(*animation: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAnimation*

Shortcut for method `aiogram.methods.send_animation.SendAnimation` will automatically fill method attributes:

- `chat_id`

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

Parameters

- **animation** – Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent animation in seconds
- **width** – Animation width
- **height** – Animation height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the animation caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the animation needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *`aiogram.methods.send_animation.SendAnimation`*

answer_audio(*audio: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, performer: Optional[str] = None, title: Optional[str] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAudio*

Shortcut for method `aiogram.methods.send_audio.SendAudio` will automatically fill method attributes:

- `chat_id`

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the `aiogram.methods.send_voice.SendVoice` method instead.

Source: https://core.telegram.org/bots/api#sendaudio

Parameters

- **audio** – Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Audio caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the audio caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving

- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_audio.SendAudio*

answer_contact(*phone_number: str*, *first_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendContact*

Shortcut for method *aiogram.methods.send_contact.SendContact* will automatically fill method attributes:

- `chat_id`

Use this method to send phone contacts. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendcontact

Parameters

- **phone_number** – Contact's phone number
- **first_name** – Contact's first name
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **last_name** – Contact's last name
- **vcard** – Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to

- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_contact.SendContact*

answer_document(*document: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, disable_content_type_detection: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendDocument*

Shortcut for method *aiogram.methods.send_document.SendDocument* will automatically fill method attributes:

- `chat_id`

Use this method to send general files. On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

Parameters

- **document** – File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the document caption. See formatting options for more details.

- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_document.SendDocument`

answer_game(*game_short_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendGame*

Shortcut for method `aiogram.methods.send_game.SendGame` will automatically fill method attributes:

- `chat_id`

Use this method to send a game. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendgame

Parameters

- **game_short_name** – Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_game.SendGame*

answer_invoice(*title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], message_thread_id: Optional[int] = None, provider_token: Optional[str] = None, max_tip_amount: Optional[int] = None, suggested_tip_amounts: Optional[List[int]] = None, start_parameter: Optional[str] = None, provider_data: Optional[str] = None, photo_url: Optional[str] = None, photo_size: Optional[int] = None, photo_width: Optional[int] = None, photo_height: Optional[int] = None, need_name: Optional[bool] = None, need_phone_number: Optional[bool] = None, need_email: Optional[bool] = None, need_shipping_address: Optional[bool] = None, send_phone_number_to_provider: Optional[bool] = None, send_email_to_provider: Optional[bool] = None, is_flexible: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendInvoice*

Shortcut for method *aiogram.methods.send_invoice.SendInvoice* will automatically fill method attributes:

- `chat_id`

Use this method to send invoices. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **currency** – Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.
- **prices** – Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **provider_token** – Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.
- **max_tip_amount** – The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts

can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.

- **start_parameter** – Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter
- **provider_data** – JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.
- **need_phone_number** – Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.
- **need_email** – Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.
- **need_shipping_address** – Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.
- **send_phone_number_to_provider** – Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.
- **send_email_to_provider** – Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.
- **is_flexible** – Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *`aiogram.methods.send_invoice.SendInvoice`*

answer_location(*latitude: float*, *longitude: float*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *live_period: Optional[int] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendLocation*

Shortcut for method `aiogram.methods.send_location.SendLocation` will automatically fill method attributes:

- `chat_id`

Use this method to send point on the map. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendlocation

Parameters

- **latitude** – Latitude of the location
- **longitude** – Longitude of the location
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **live_period** – Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.
- **heading** – For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_location.SendLocation*

answer_media_group(*media: List[Union[InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMediaGroup*

Shortcut for method *aiogram.methods.send_media_group.SendMediaGroup* will automatically fill method attributes:

- `chat_id`

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **media** – A JSON-serialized array describing messages to be sent, must include 2-10 items
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends messages silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent messages from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the messages are a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_media_group.SendMediaGroup*

answer_photo(*photo: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPhoto*

Shortcut for method `aiogram.methods.send_photo.SendPhoto` will automatically fill method attributes:

- `chat_id`

Use this method to send photos. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendphoto

Parameters

- **photo** – Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the photo caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the photo needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_photo.SendPhoto`

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

answer_poll(*question: str, options: List[Union[InputPollOption, str]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, question_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, question_entities: Optional[List[MessageEntity]] = None, is_anonymous: Optional[bool] = None, type: Optional[str] = None, allows_multiple_answers: Optional[bool] = None, correct_option_id: Optional[int] = None, explanation: Optional[str] = None, explanation_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, explanation_entities: Optional[List[MessageEntity]] = None, open_period: Optional[int] = None, close_date: Optional[Union[datetime.datetime, datetime.timedelta, int]] = None, is_closed: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → SendPoll

Shortcut for method `aiogram.methods.send_poll.SendPoll` will automatically fill method attributes:

- `chat_id`

Use this method to send a native poll. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpoll

Parameters

- **question** – Poll question, 1-300 characters
- **options** – A JSON-serialized list of 2-10 answer options
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **question_parse_mode** – Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed
- **question_entities** – A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*
- **is_anonymous** – `True`, if the poll needs to be anonymous, defaults to `True`
- **type** – Poll type, 'quiz' or 'regular', defaults to 'regular'
- **allows_multiple_answers** – `True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`

- **correct_option_id** – 0-based identifier of the correct answer option, required for polls in quiz mode
- **explanation** – Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing
- **explanation_parse_mode** – Mode for parsing entities in the explanation. See formatting options for more details.
- **explanation_entities** – A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*
- **open_period** – Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.
- **close_date** – Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.
- **is_closed** – Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_poll.SendPoll*

answer_dice(*business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendDice*

Shortcut for method *aiogram.methods.send_dice.SendDice* will automatically fill method attributes:

- `chat_id`

Use this method to send an animated emoji that will display a random value. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#senddice

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_dice.SendDice*

answer_sticker(*sticker: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendSticker*

Shortcut for method *aiogram.methods.send_sticker.SendSticker* will automatically fill method attributes:

- `chat_id`

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendsticker

Parameters

- **sticker** – Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

- **emoji** – Emoji associated with the sticker; only for just uploaded stickers
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_sticker.SendSticker`

answer_venue(*latitude: float*, *longitude: float*, *title: str*, *address: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendVenue*

Shortcut for method `aiogram.methods.send_venue.SendVenue` will automatically fill method attributes:

- `chat_id`

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

Parameters

- **latitude** – Latitude of the venue
- **longitude** – Longitude of the venue
- **title** – Name of the venue
- **address** – Address of the venue
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **foursquare_id** – Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)
- **google_place_id** – Google Places identifier of the venue

- **google_place_type** – Google Places type of the venue. (See supported types.)
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_venue.SendVenue`

answer_video(*video: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, supports_streaming: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideo*

Shortcut for method `aiogram.methods.send_video.SendVideo` will automatically fill method attributes:

- `chat_id`

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

Parameters

- **video** – Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the video needs to be covered with a spoiler animation
- **supports_streaming** – Pass `True` if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_video.SendVideo`

answer_video_note(*video_note: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, length: Optional[int] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideoNote*

Shortcut for method *aiogram.methods.send_video_note.SendVideoNote* will automatically fill method attributes:

- `chat_id`

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **video_note** – Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **length** – Video width and height, i.e. diameter of the video message
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_video_note.SendVideoNote*

answer_voice(*voice: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVoice*

Shortcut for method `aiogram.methods.send_voice.SendVoice` will automatically fill method attributes:

- `chat_id`

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as `aiogram.types.audio.Audio` or `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

Parameters

- **voice** – Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Voice message caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the voice message caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the voice message in seconds
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

- `reply_to_message_id` – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_voice.SendVoice`

ChatPermissions

`class aiogram.types.chat_permissions.ChatPermissions`(**, can_send_messages: bool | None = None, can_send_audios: bool | None = None, can_send_documents: bool | None = None, can_send_photos: bool | None = None, can_send_videos: bool | None = None, can_send_video_notes: bool | None = None, can_send_voice_notes: bool | None = None, can_send_polls: bool | None = None, can_send_other_messages: bool | None = None, can_add_web_page_previews: bool | None = None, can_change_info: bool | None = None, can_invite_users: bool | None = None, can_pin_messages: bool | None = None, can_manage_topics: bool | None = None, **extra_data: Any*)

Describes actions that a non-administrator user is allowed to take in a chat.

Source: https://core.telegram.org/bots/api#chatpermissions

`can_send_messages: bool | None`

Optional. `True`, if the user is allowed to send text messages, contacts, giveaways, giveaway winners, invoices, locations and venues

`can_send_audios: bool | None`

Optional. `True`, if the user is allowed to send audios

`can_send_documents: bool | None`

Optional. `True`, if the user is allowed to send documents

`can_send_photos: bool | None`

Optional. `True`, if the user is allowed to send photos

`can_send_videos: bool | None`

Optional. `True`, if the user is allowed to send videos

`can_send_video_notes: bool | None`

Optional. `True`, if the user is allowed to send video notes

`can_send_voice_notes: bool | None`

Optional. `True`, if the user is allowed to send voice notes

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`can_send_polls: bool | None`

Optional. `True`, if the user is allowed to send polls

can_send_other_messages: bool | None

Optional. True, if the user is allowed to send animations, games, stickers and use inline bots

can_add_web_page_previews: bool | None

Optional. True, if the user is allowed to add web page previews to their messages

can_change_info: bool | None

Optional. True, if the user is allowed to change the chat title, photo and other settings. Ignored in public supergroups

can_invite_users: bool | None

Optional. True, if the user is allowed to invite new users to the chat

can_pin_messages: bool | None

Optional. True, if the user is allowed to pin messages. Ignored in public supergroups

can_manage_topics: bool | None

Optional. True, if the user is allowed to create forum topics. If omitted defaults to the value of can_pin_messages

ChatPhoto

class aiogram.types.chat_photo.**ChatPhoto**(*, *small_file_id: str*, *small_file_unique_id: str*, *big_file_id: str*, *big_file_unique_id: str*, ***extra_data: Any*)

This object represents a chat photo.

Source: https://core.telegram.org/bots/api#chatphoto

small_file_id: str

File identifier of small (160x160) chat photo. This file_id can be used only for photo download and only for as long as the photo is not changed.

small_file_unique_id: str

Unique file identifier of small (160x160) chat photo, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

big_file_id: str

File identifier of big (640x640) chat photo. This file_id can be used only for photo download and only for as long as the photo is not changed.

big_file_unique_id: str

Unique file identifier of big (640x640) chat photo, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

ChatShared

class `aiogram.types.chat_shared.`**`ChatShared`**(*, *request_id: int*, *chat_id: int*, *title: str | None = None*, *username: str | None = None*, *photo: List[PhotoSize] | None = None*, ***extra_data: Any*)

This object contains information about a chat that was shared with the bot using a `aiogram.types.keyboard_button_request_chat.KeyboardButtonRequestChat` button.

Source: https://core.telegram.org/bots/api#chatshared

`request_id: int`

Identifier of the request

`chat_id: int`

Identifier of the shared chat. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a 64-bit integer or double-precision float type are safe for storing this identifier. The bot may not have access to the chat and could be unable to use this identifier, unless the chat is already known to the bot by some other means.

`title: str | None`

Optional. Title of the chat, if the title was requested by the bot.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`username: str | None`

Optional. Username of the chat, if the username was requested by the bot and available.

`photo: List[PhotoSize] | None`

Optional. Available sizes of the chat photo, if the photo was requested by the bot

Contact

class `aiogram.types.contact.`**`Contact`**(*, *phone_number: str*, *first_name: str*, *last_name: str | None = None*, *user_id: int | None = None*, *vcard: str | None = None*, ***extra_data: Any*)

This object represents a phone contact.

Source: https://core.telegram.org/bots/api#contact

`phone_number: str`

Contact's phone number

`first_name: str`

Contact's first name

`last_name: str | None`

Optional. Contact's last name

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_id: int | None

Optional. Contact's user identifier in Telegram. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a 64-bit integer or double-precision float type are safe for storing this identifier.

vcard: str | None

Optional. Additional data about the contact in the form of a vCard

Dice

class aiogram.types.dice.**Dice**(***, *emoji: str*, *value: int*, ***extra_data: Any*)

This object represents an animated emoji that displays a random value.

Source: https://core.telegram.org/bots/api#dice

emoji: str

Emoji on which the dice throw animation is based

value: int

Value of the dice, 1-6 for '', '' and '' base emoji, 1-5 for '' and '' base emoji, 1-64 for '' base emoji

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class aiogram.types.dice.**DiceEmoji**

DICE = ''

DART = ''

BASKETBALL = ''

FOOTBALL = ''

SLOT_MACHINE = ''

BOWLING = ''

Document

class aiogram.types.document.**Document**(***, *file_id: str*, *file_unique_id: str*, *thumbnail: PhotoSize | None = None*, *file_name: str | None = None*, *mime_type: str | None = None*, *file_size: int | None = None*, ***extra_data: Any*)

This object represents a general file (as opposed to photos, voice messages and audio files).

Source: https://core.telegram.org/bots/api#document

file_id: str

Identifier for this file, which can be used to download or reuse the file

file_unique_id: str

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

thumbnail: *PhotoSize* | None

Optional. Document thumbnail as defined by the sender

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_name: str | None

Optional. Original filename as defined by the sender

mime_type: str | None

Optional. MIME type of the file as defined by the sender

file_size: int | None

Optional. File size in bytes. It can be bigger than 2^31 and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this value.

ExternalReplyInfo

class aiogram.types.external_reply_info.**ExternalReplyInfo**(**, origin: MessageOriginUser | MessageOriginHiddenUser | MessageOriginChat | MessageOriginChannel, chat: Chat | None = None, message_id: int | None = None, link_preview_options: LinkPreviewOptions | None = None, animation: Animation | None = None, audio: Audio | None = None, document: Document | None = None, paid_media: PaidMediaInfo | None = None, photo: List[PhotoSize] | None = None, sticker: Sticker | None = None, story: Story | None = None, video: Video | None = None, video_note: VideoNote | None = None, voice: Voice | None = None, has_media_spoiler: bool | None = None, contact: Contact | None = None, dice: Dice | None = None, game: Game | None = None, giveaway: Giveaway | None = None, giveaway_winners: GiveawayWinners | None = None, invoice: Invoice | None = None, location: Location | None = None, poll: Poll | None = None, venue: Venue | None = None, **extra_data: Any*)

This object contains information about a message that is being replied to, which may come from another chat or forum topic.

Source: https://core.telegram.org/bots/api#externalreplyinfo

origin: *MessageOriginUser* | *MessageOriginHiddenUser* | *MessageOriginChat* | *MessageOriginChannel*

Origin of the message replied to by the given message

chat: *Chat* | None

Optional. Chat the original message belongs to. Available only if the chat is a supergroup or a channel.

message_id: int | None

Optional. Unique message identifier inside the original chat. Available only if the original chat is a supergroup or a channel.

link_preview_options: *LinkPreviewOptions* | None

Optional. Options used for link preview generation for the original message, if it is a text message

animation: *Animation* | None

Optional. Message is an animation, information about the animation

audio: *Audio* | None

Optional. Message is an audio file, information about the file

document: *Document* | None

Optional. Message is a general file, information about the file

paid_media: *PaidMediaInfo* | None

Optional. Message contains paid media; information about the paid media

photo: List[*PhotoSize*] | None

Optional. Message is a photo, available sizes of the photo

sticker: *Sticker* | None

Optional. Message is a sticker, information about the sticker

story: *Story* | None

Optional. Message is a forwarded story

video: *Video* | None

Optional. Message is a video, information about the video

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

video_note: *VideoNote* | None

Optional. Message is a video note, information about the video message

voice: *Voice* | None

Optional. Message is a voice message, information about the file

has_media_spoiler: bool | None

Optional. `True`, if the message media is covered by a spoiler animation

contact: *Contact* | None

Optional. Message is a shared contact, information about the contact

dice: *Dice* **| None**

Optional. Message is a dice with random value

game: *Game* **| None**

Optional. Message is a game, information about the game. More about games »

giveaway: *Giveaway* **| None**

Optional. Message is a scheduled giveaway, information about the giveaway

giveaway_winners: *GiveawayWinners* **| None**

Optional. A giveaway with public winners was completed

invoice: *Invoice* **| None**

Optional. Message is an invoice for a payment, information about the invoice. More about payments »

location: *Location* **| None**

Optional. Message is a shared location, information about the location

poll: *Poll* **| None**

Optional. Message is a native poll, information about the poll

venue: *Venue* **| None**

Optional. Message is a venue, information about the venue

File

class aiogram.types.file.**File**(**, file_id: str, file_unique_id: str, file_size: int | None = None, file_path: str | None = None, **extra_data: Any*)

This object represents a file ready to be downloaded. The file can be downloaded via the link `https://api.telegram.org/file/bot<token>/<file_path>`. It is guaranteed that the link will be valid for at least 1 hour. When the link expires, a new one can be requested by calling *aiogram.methods.get_file.GetFile*.

The maximum file size to download is 20 MB

Source: https://core.telegram.org/bots/api#file

file_id: str

Identifier for this file, which can be used to download or reuse the file

file_unique_id: str

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_size: int | None

Optional. File size in bytes. It can be bigger than 2^31 and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this value.

file_path: str | None

Optional. File path. Use `https://api.telegram.org/file/bot<token>/<file_path>` to get the file.

ForceReply

`class aiogram.types.force_reply.ForceReply`(**, force_reply: Literal[True] = True, input_field_placeholder: str | None = None, selective: bool | None = None, **extra_data: Any*)

Upon receiving a message with this object, Telegram clients will display a reply interface to the user (act as if the user has selected the bot's message and tapped 'Reply'). This can be extremely useful if you want to create user-friendly step-by-step interfaces without having to sacrifice privacy mode. Not supported in channels and for messages sent on behalf of a Telegram Business account.

> **Example:** A poll bot for groups runs in privacy mode (only receives commands, replies to its messages and mentions). There could be two ways to create a new poll:
>
> - Explain the user how to send a command with parameters (e.g. /newpoll question answer1 answer2). May be appealing for hardcore users but lacks modern day polish.
> - Guide the user through a step-by-step process. 'Please send me your question', 'Cool, now let's add the first answer option', 'Great. Keep adding answer options, then send /done when you're ready'.
>
> The last option is definitely more attractive. And if you use *`aiogram.types.force_reply.ForceReply`* in your bot's questions, it will receive the user's answers even if it only receives replies, commands and mentions - without any extra work for the user.

Source: https://core.telegram.org/bots/api#forcereply

`force_reply: Literal[True]`

Shows reply interface to the user, as if they manually selected the bot's message and tapped 'Reply'

`input_field_placeholder: str | None`

Optional. The placeholder to be shown in the input field when the reply is active; 1-64 characters

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`selective: bool | None`

Optional. Use this parameter if you want to force reply from specific users only. Targets: 1) users that are @mentioned in the *text* of the *`aiogram.types.message.Message`* object; 2) if the bot's message is a reply to a message in the same chat and forum topic, sender of the original message.

ForumTopic

`class aiogram.types.forum_topic.ForumTopic`(**, message_thread_id: int, name: str, icon_color: int, icon_custom_emoji_id: str | None = None, **extra_data: Any*)

This object represents a forum topic.

Source: https://core.telegram.org/bots/api#forumtopic

`message_thread_id: int`

Unique identifier of the forum topic

name: str

Name of the topic

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

icon_color: int

Color of the topic icon in RGB format

icon_custom_emoji_id: str | None

Optional. Unique identifier of the custom emoji shown as the topic icon

ForumTopicClosed

class aiogram.types.forum_topic_closed.**ForumTopicClosed**(***extra_data: Any*)

This object represents a service message about a forum topic closed in the chat. Currently holds no information.

Source: https://core.telegram.org/bots/api#forumtopicclosed

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ForumTopicCreated

class aiogram.types.forum_topic_created.**ForumTopicCreated**(**, *name: str*, *icon_color: int*, *icon_custom_emoji_id: str | None = None*, ***extra_data: Any*)

This object represents a service message about a new forum topic created in the chat.

Source: https://core.telegram.org/bots/api#forumtopiccreated

name: str

Name of the topic

icon_color: int

Color of the topic icon in RGB format

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

icon_custom_emoji_id: str | None

Optional. Unique identifier of the custom emoji shown as the topic icon

ForumTopicEdited

class aiogram.types.forum_topic_edited.**ForumTopicEdited**(*, *name: str | None = None*, *icon_custom_emoji_id: str | None = None*, ***extra_data: Any*)

This object represents a service message about an edited forum topic.

Source: https://core.telegram.org/bots/api#forumtopicedited

name: str | None

Optional. New name of the topic, if it was edited

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

icon_custom_emoji_id: str | None

Optional. New identifier of the custom emoji shown as the topic icon, if it was edited; an empty string if the icon was removed

ForumTopicReopened

class aiogram.types.forum_topic_reopened.**ForumTopicReopened**(***extra_data: Any*)

This object represents a service message about a forum topic reopened in the chat. Currently holds no information.

Source: https://core.telegram.org/bots/api#forumtopicreopened

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

GeneralForumTopicHidden

class aiogram.types.general_forum_topic_hidden.**GeneralForumTopicHidden**(***extra_data: Any*)

This object represents a service message about General forum topic hidden in the chat. Currently holds no information.

Source: https://core.telegram.org/bots/api#generalforumtopichidden

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

GeneralForumTopicUnhidden

class `aiogram.types.general_forum_topic_unhidden.`**`GeneralForumTopicUnhidden`**(***extra_data: Any*)

This object represents a service message about General forum topic unhidden in the chat. Currently holds no information.

Source: https://core.telegram.org/bots/api#generalforumtopicunhidden

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Giveaway

class `aiogram.types.giveaway.`**`Giveaway`**(**, chats: List[Chat], winners_selection_date: _datetime_serializer, return_type=int, when_used=unless - none)], winner_count: int, only_new_members: bool | None = None, has_public_winners: bool | None = None, prize_description: str | None = None, country_codes: List[str] | None = None, prize_star_count: int | None = None, premium_subscription_month_count: int | None = None, **extra_data: Any*)

This object represents a message about a scheduled giveaway.

Source: https://core.telegram.org/bots/api#giveaway

`chats: List[Chat]`

The list of chats which the user must join to participate in the giveaway

`winners_selection_date: DateTime`

Point in time (Unix timestamp) when winners of the giveaway will be selected

`winner_count: int`

The number of users which are supposed to be selected as winners of the giveaway

`only_new_members: bool | None`

Optional. `True`, if only users who join the chats after the giveaway started should be eligible to win

`has_public_winners: bool | None`

Optional. `True`, if the list of giveaway winners will be visible to everyone

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`prize_description: str | None`

Optional. Description of additional giveaway prize

`country_codes: List[str] | None`

Optional. A list of two-letter ISO 3166-1 alpha-2 country codes indicating the countries from which eligible users for the giveaway must come. If empty, then all users can participate in the giveaway. Users with a phone number that was bought on Fragment can always participate in giveaways.

prize_star_count: int | None

Optional. The number of Telegram Stars to be split between giveaway winners; for Telegram Star giveaways only

premium_subscription_month_count: int | None

Optional. The number of months the Telegram Premium subscription won from the giveaway will be active for; for Telegram Premium giveaways only

GiveawayCompleted

class aiogram.types.giveaway_completed.**GiveawayCompleted**(*, *winner_count: int*, *unclaimed_prize_count: int | None = None*, *giveaway_message: Message | None = None*, *is_star_giveaway: bool | None = None*, ***extra_data: Any*)

This object represents a service message about the completion of a giveaway without public winners.

Source: https://core.telegram.org/bots/api#giveawaycompleted

winner_count: int

Number of winners in the giveaway

unclaimed_prize_count: int | None

Optional. Number of undistributed prizes

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

giveaway_message: *Message* | None

Optional. Message with the giveaway that was completed, if it wasn't deleted

is_star_giveaway: bool | None

Optional. `True`, if the giveaway is a Telegram Star giveaway. Otherwise, currently, the giveaway is a Telegram Premium giveaway.

GiveawayCreated

class aiogram.types.giveaway_created.**GiveawayCreated**(*, *prize_star_count: int | None = None*, ***extra_data: Any*)

This object represents a service message about the creation of a scheduled giveaway.

Source: https://core.telegram.org/bots/api#giveawaycreated

prize_star_count: int | None

Optional. The number of Telegram Stars to be split between giveaway winners; for Telegram Star giveaways only

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

GiveawayWinners

class aiogram.types.giveaway_winners.**GiveawayWinners**(***, *chat: Chat*, *giveaway_message_id: int*, *winners_selection_date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *winner_count: int*, *winners: List[User]*, *additional_chat_count: int | None = None*, *prize_star_count: int | None = None*, *premium_subscription_month_count: int | None = None*, *unclaimed_prize_count: int | None = None*, *only_new_members: bool | None = None*, *was_refunded: bool | None = None*, *prize_description: str | None = None*, ***extra_data: Any*)

This object represents a message about the completion of a giveaway with public winners.

Source: https://core.telegram.org/bots/api#giveawaywinners

chat: *Chat*

The chat that created the giveaway

giveaway_message_id: int

Identifier of the message with the giveaway in the chat

winners_selection_date: DateTime

Point in time (Unix timestamp) when winners of the giveaway were selected

winner_count: int

Total number of winners in the giveaway

winners: List[*User*]

List of up to 100 winners of the giveaway

additional_chat_count: int | None

Optional. The number of other chats the user had to join in order to be eligible for the giveaway

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

prize_star_count: int | None

Optional. The number of Telegram Stars that were split between giveaway winners; for Telegram Star giveaways only

premium_subscription_month_count: int | None

Optional. The number of months the Telegram Premium subscription won from the giveaway will be active for; for Telegram Premium giveaways only

unclaimed_prize_count: int | None

Optional. Number of undistributed prizes

only_new_members: bool | None

Optional. `True`, if only users who had joined the chats after the giveaway started were eligible to win

was_refunded: bool | None

Optional. True, if the giveaway was canceled because the payment for it was refunded

prize_description: str | None

Optional. Description of additional giveaway prize

InaccessibleMessage

class aiogram.types.inaccessible_message.**InaccessibleMessage**(**, chat: Chat, message_id: int, date: Literal[0] = 0, **extra_data: Any*)

This object describes a message that was deleted or is otherwise inaccessible to the bot.

Source: https://core.telegram.org/bots/api#inaccessiblemessage

chat: *Chat*

Chat the message belonged to

message_id: int

Unique message identifier inside the chat

date: Literal[0]

Always 0. The field can be used to differentiate regular and inaccessible messages.

answer(*text: str, business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, entities: Optional[List[MessageEntity]] = None, link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMessage*

Shortcut for method *aiogram.methods.send_message.SendMessage* will automatically fill method attributes:

- chat_id

Use this method to send text messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendmessage

Parameters

- **text** – Text of the message to be sent, 1-4096 characters after entities parsing
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **parse_mode** – Mode for parsing entities in the message text. See formatting options for more details.
- **entities** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*

- **link_preview_options** – Link preview generation options for the message
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **disable_web_page_preview** – Disables link previews for links in this message
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_message.SendMessage*

reply(*text: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>*, *entities: Optional[List[MessageEntity]] = None*, *link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>*, ***kwargs: Any*) → *SendMessage*

Shortcut for method *aiogram.methods.send_message.SendMessage* will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send text messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendmessage

Parameters

- **text** – Text of the message to be sent, 1-4096 characters after entities parsing
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **parse_mode** – Mode for parsing entities in the message text. See formatting options for more details.
- **entities** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*
- **link_preview_options** – Link preview generation options for the message

- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **disable_web_page_preview** – Disables link previews for links in this message

Returns
instance of method *aiogram.methods.send_message.SendMessage*

answer_animation(*animation: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAnimation*

Shortcut for method *aiogram.methods.send_animation.SendAnimation* will automatically fill method attributes:

- `chat_id`

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

Parameters

- **animation** – Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent animation in seconds
- **width** – Animation width

- **height** – Animation height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the animation caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the animation needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_animation.SendAnimation`

reply_animation(*animation: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendAnimation*

Shortcut for method `aiogram.methods.send_animation.SendAnimation` will automatically fill method attributes:

- `chat_id`

- `reply_parameters`

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

Parameters

- **animation** – Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent animation in seconds
- **width** – Animation width
- **height** – Animation height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the animation caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the animation needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_animation.SendAnimation`

answer_audio(*audio: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, performer: Optional[str] = None, title: Optional[str] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAudio*

Shortcut for method `aiogram.methods.send_audio.SendAudio` will automatically fill method attributes:

- `chat_id`

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the `aiogram.methods.send_voice.SendVoice` method instead.

Source: https://core.telegram.org/bots/api#sendaudio

Parameters

- **audio** – Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Audio caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the audio caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *`aiogram.methods.send_audio.SendAudio`*

reply_audio(*audio: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, performer: Optional[str] = None, title: Optional[str] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendAudio*

Shortcut for method *`aiogram.methods.send_audio.SendAudio`* will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent *`aiogram.types.message.Message`* is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the *`aiogram.methods.send_voice.SendVoice`* method instead.

Source: https://core.telegram.org/bots/api#sendaudio

Parameters

- **audio** – Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Audio caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the audio caption. See formatting options for more details.

- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_audio.SendAudio`

answer_contact(*phone_number: str*, *first_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendContact*

Shortcut for method `aiogram.methods.send_contact.SendContact` will automatically fill method attributes:

- `chat_id`

Use this method to send phone contacts. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendcontact

Parameters

- **phone_number** – Contact's phone number
- **first_name** – Contact's first name
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent

- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **last_name** – Contact's last name
- **vcard** – Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_contact.SendContact*

reply_contact(*phone_number: str*, *first_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → *SendContact*

Shortcut for method *aiogram.methods.send_contact.SendContact* will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send phone contacts. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendcontact

Parameters

- **phone_number** – Contact's phone number
- **first_name** – Contact's first name
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **last_name** – Contact's last name
- **vcard** – Additional data about the contact in the form of a vCard, 0-2048 bytes

- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_contact.SendContact`

answer_document(*document: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, disable_content_type_detection: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendDocument*

Shortcut for method `aiogram.methods.send_document.SendDocument` will automatically fill method attributes:

- `chat_id`

Use this method to send general files. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

Parameters

- **document** – File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

- **caption** – Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the document caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_document.SendDocument`

reply_document(*document: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, disable_content_type_detection: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendDocument*

Shortcut for method `aiogram.methods.send_document.SendDocument` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send general files. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

Parameters

- **document** – File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the document caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_document.SendDocument`

answer_game(*game_short_name: str, business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendGame*

Shortcut for method `aiogram.methods.send_game.SendGame` will automatically fill method attributes:

- `chat_id`

Use this method to send a game. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendgame

Parameters

- **game_short_name** – Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_game.SendGame`

reply_game(*game_short_name: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → SendGame

Shortcut for method `aiogram.methods.send_game.SendGame` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send a game. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendgame

Parameters

- **game_short_name** – Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method *aiogram.methods.send_game.SendGame*

answer_invoice(*title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], message_thread_id: Optional[int] = None, provider_token: Optional[str] = None, max_tip_amount: Optional[int] = None, suggested_tip_amounts: Optional[List[int]] = None, start_parameter: Optional[str] = None, provider_data: Optional[str] = None, photo_url: Optional[str] = None, photo_size: Optional[int] = None, photo_width: Optional[int] = None, photo_height: Optional[int] = None, need_name: Optional[bool] = None, need_phone_number: Optional[bool] = None, need_email: Optional[bool] = None, need_shipping_address: Optional[bool] = None, send_phone_number_to_provider: Optional[bool] = None, send_email_to_provider: Optional[bool] = None, is_flexible: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendInvoice*

Shortcut for method *aiogram.methods.send_invoice.SendInvoice* will automatically fill method attributes:

- `chat_id`

Use this method to send invoices. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **currency** – Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.
- **prices** – Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **provider_token** – Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.
- **max_tip_amount** – The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts

can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.

- **start_parameter** – Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter
- **provider_data** – JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.
- **need_phone_number** – Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.
- **need_email** – Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.
- **need_shipping_address** – Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.
- **send_phone_number_to_provider** – Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.
- **send_email_to_provider** – Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.
- **is_flexible** – Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_invoice.SendInvoice*

reply_invoice(*title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], message_thread_id: Optional[int] = None, provider_token: Optional[str] = None, max_tip_amount: Optional[int] = None, suggested_tip_amounts: Optional[List[int]] = None, start_parameter: Optional[str] = None, provider_data: Optional[str] = None, photo_url: Optional[str] = None, photo_size: Optional[int] = None, photo_width: Optional[int] = None, photo_height: Optional[int] = None, need_name: Optional[bool] = None, need_phone_number: Optional[bool] = None, need_email: Optional[bool] = None, need_shipping_address: Optional[bool] = None, send_phone_number_to_provider: Optional[bool] = None, send_email_to_provider: Optional[bool] = None, is_flexible: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendInvoice*

Shortcut for method `aiogram.methods.send_invoice.SendInvoice` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send invoices. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **currency** – Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.
- **prices** – Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **provider_token** – Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.
- **max_tip_amount** – The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.
- **start_parameter** – Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent

message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter

- **provider_data** – JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.
- **need_phone_number** – Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.
- **need_email** – Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.
- **need_shipping_address** – Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.
- **send_phone_number_to_provider** – Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.
- **send_email_to_provider** – Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.
- **is_flexible** – Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method *aiogram.methods.send_invoice.SendInvoice*

answer_location(*latitude: float*, *longitude: float*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *live_period: Optional[int] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendLocation*

Shortcut for method *aiogram.methods.send_location.SendLocation* will automatically fill method attributes:

- chat_id

Use this method to send point on the map. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendlocation

Parameters

- **latitude** – Latitude of the location
- **longitude** – Longitude of the location
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **live_period** – Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.
- **heading** – For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass True if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_location.SendLocation*

reply_location(*latitude: float*, *longitude: float*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *live_period: Optional[int] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → *SendLocation*

Shortcut for method `aiogram.methods.send_location.SendLocation` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send point on the map. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendlocation

Parameters

- **latitude** – Latitude of the location
- **longitude** – Longitude of the location
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **live_period** – Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.
- **heading** – For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_location.SendLocation`

answer_media_group(*media: List[Union[InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMediaGroup*

Shortcut for method `aiogram.methods.send_media_group.SendMediaGroup` will automatically fill method attributes:

- `chat_id`

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **media** – A JSON-serialized array describing messages to be sent, must include 2-10 items
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends messages silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent messages from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the messages are a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_media_group.SendMediaGroup`

reply_media_group(*media: List[Union[InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendMediaGroup*

Shortcut for method `aiogram.methods.send_media_group.SendMediaGroup` will automatically fill method attributes:

- `chat_id`

- `reply_parameters`

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **media** – A JSON-serialized array describing messages to be sent, must include 2-10 items
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends messages silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent messages from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method `aiogram.methods.send_media_group.SendMediaGroup`

answer_photo(*photo: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPhoto*

Shortcut for method `aiogram.methods.send_photo.SendPhoto` will automatically fill method attributes:

- `chat_id`

Use this method to send photos. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendphoto

Parameters

- **photo** – Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent

- **`message_thread_id`** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **`caption`** – Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing
- **`parse_mode`** – Mode for parsing entities in the photo caption. See formatting options for more details.
- **`caption_entities`** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **`show_caption_above_media`** – Pass `True`, if the caption must be shown above the message media
- **`has_spoiler`** – Pass `True` if the photo needs to be covered with a spoiler animation
- **`disable_notification`** – Sends the message silently. Users will receive a notification with no sound.
- **`protect_content`** – Protects the contents of the sent message from forwarding and saving
- **`message_effect_id`** – Unique identifier of the message effect to be added to the message; for private chats only
- **`reply_parameters`** – Description of the message to reply to
- **`reply_markup`** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **`allow_sending_without_reply`** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **`reply_to_message_id`** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_photo.SendPhoto`

`reply_photo`(*photo: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendPhoto*

Shortcut for method `aiogram.methods.send_photo.SendPhoto` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send photos. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendphoto

Parameters

- **photo** – Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the photo caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the photo needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_photo.SendPhoto`

answer_poll(*question: str, options: List[Union[InputPollOption, str]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, question_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, question_entities: Optional[List[MessageEntity]] = None, is_anonymous: Optional[bool] = None, type: Optional[str] = None, allows_multiple_answers: Optional[bool] = None, correct_option_id: Optional[int] = None, explanation: Optional[str] = None, explanation_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, explanation_entities: Optional[List[MessageEntity]] = None, open_period: Optional[int] = None, close_date: Optional[Union[datetime.datetime, datetime.timedelta, int]] = None, is_closed: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPoll*

Shortcut for method `aiogram.methods.send_poll.SendPoll` will automatically fill method attributes:

- `chat_id`

Use this method to send a native poll. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpoll

Parameters

- **question** – Poll question, 1-300 characters
- **options** – A JSON-serialized list of 2-10 answer options
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **question_parse_mode** – Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed
- **question_entities** – A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*
- **is_anonymous** – `True`, if the poll needs to be anonymous, defaults to `True`
- **type** – Poll type, 'quiz' or 'regular', defaults to 'regular'
- **allows_multiple_answers** – `True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`
- **correct_option_id** – 0-based identifier of the correct answer option, required for polls in quiz mode
- **explanation** – Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing
- **explanation_parse_mode** – Mode for parsing entities in the explanation. See formatting options for more details.
- **explanation_entities** – A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*
- **open_period** – Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.
- **close_date** – Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.
- **is_closed** – Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to

- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_poll.SendPoll`

reply_poll(*question: str, options: List[Union[InputPollOption, str]], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, question_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, question_entities: Optional[List[MessageEntity]] = None, is_anonymous: Optional[bool] = None, type: Optional[str] = None, allows_multiple_answers: Optional[bool] = None, correct_option_id: Optional[int] = None, explanation: Optional[str] = None, explanation_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, explanation_entities: Optional[List[MessageEntity]] = None, open_period: Optional[int] = None, close_date: Optional[Union[datetime.datetime, datetime.timedelta, int]] = None, is_closed: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendPoll*

Shortcut for method `aiogram.methods.send_poll.SendPoll` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send a native poll. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpoll

Parameters

- **question** – Poll question, 1-300 characters
- **options** – A JSON-serialized list of 2-10 answer options
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **question_parse_mode** – Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed
- **question_entities** – A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*
- **is_anonymous** – `True`, if the poll needs to be anonymous, defaults to `True`
- **type** – Poll type, 'quiz' or 'regular', defaults to 'regular'
- **allows_multiple_answers** – `True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`

- **correct_option_id** – 0-based identifier of the correct answer option, required for polls in quiz mode
- **explanation** – Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing
- **explanation_parse_mode** – Mode for parsing entities in the explanation. See formatting options for more details.
- **explanation_entities** – A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*
- **open_period** – Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.
- **close_date** – Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.
- **is_closed** – Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method *aiogram.methods.send_poll.SendPoll*

answer_dice(*business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *emoji: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendDice*

Shortcut for method *aiogram.methods.send_dice.SendDice* will automatically fill method attributes:

- `chat_id`

Use this method to send an animated emoji that will display a random value. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#senddice

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent

- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_dice.SendDice`

reply_dice(*business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → SendDice

Shortcut for method `aiogram.methods.send_dice.SendDice` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send an animated emoji that will display a random value. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#senddice

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding

- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_dice.SendDice`

answer_sticker(*sticker: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendSticker*

Shortcut for method `aiogram.methods.send_sticker.SendSticker` will automatically fill method attributes:

- `chat_id`

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendsticker

Parameters

- **sticker** – Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji associated with the sticker; only for just uploaded stickers
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_sticker.SendSticker*

reply_sticker(*sticker: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendSticker*

Shortcut for method *aiogram.methods.send_sticker.SendSticker* will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendsticker

Parameters

- **sticker** – Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **emoji** – Emoji associated with the sticker; only for just uploaded stickers
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method *aiogram.methods.send_sticker.SendSticker*

answer_venue(*latitude: float*, *longitude: float*, *title: str*, *address: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendVenue*

Shortcut for method `aiogram.methods.send_venue.SendVenue` will automatically fill method attributes:

- `chat_id`

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

Parameters

- **latitude** – Latitude of the venue
- **longitude** – Longitude of the venue
- **title** – Name of the venue
- **address** – Address of the venue
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **foursquare_id** – Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)
- **google_place_id** – Google Places identifier of the venue
- **google_place_type** – Google Places type of the venue. (See supported types.)
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_venue.SendVenue`

reply_venue(*latitude: float*, *longitude: float*, *title: str*, *address: str*, *business_connection_id: Optional[str] = None*, *message_thread_id: Optional[int] = None*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → *SendVenue*

Shortcut for method `aiogram.methods.send_venue.SendVenue` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

Parameters

- **latitude** – Latitude of the venue
- **longitude** – Longitude of the venue
- **title** – Name of the venue
- **address** – Address of the venue
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **foursquare_id** – Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)
- **google_place_id** – Google Places identifier of the venue
- **google_place_type** – Google Places type of the venue. (See supported types.)
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method `aiogram.methods.send_venue.SendVenue`

answer_video(*video: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, supports_streaming: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideo*

Shortcut for method `aiogram.methods.send_video.SendVideo` will automatically fill method attributes:

- `chat_id`

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

Parameters

- **video** – Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the video needs to be covered with a spoiler animation
- **supports_streaming** – Pass `True` if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_video.SendVideo`

reply_video(*video: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, supports_streaming: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendVideo*

Shortcut for method `aiogram.methods.send_video.SendVideo` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

Parameters

- **video** – Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the video needs to be covered with a spoiler animation
- **supports_streaming** – Pass `True` if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_video.SendVideo`

answer_video_note(*video_note: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, length: Optional[int] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideoNote*

Shortcut for method *aiogram.methods.send_video_note.SendVideoNote* will automatically fill method attributes:

- chat_id

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **video_note** – Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **length** – Video width and height, i.e. diameter of the video message
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass True if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_video_note.SendVideoNote*

reply_video_note(*video_note: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, duration: Optional[int] = None, length: Optional[int] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendVideoNote*

Shortcut for method `aiogram.methods.send_video_note.SendVideoNote` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **video_note** – Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **duration** – Duration of sent video in seconds
- **length** – Video width and height, i.e. diameter of the video message
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method `aiogram.methods.send_video_note.SendVideoNote`

answer_voice(*voice: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVoice*

Shortcut for method `aiogram.methods.send_voice.SendVoice` will automatically fill method attributes:

- `chat_id`

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as `aiogram.types.audio.Audio` or `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

Parameters

- **voice** – Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Voice message caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the voice message caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the voice message in seconds
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_voice.SendVoice*

reply_voice(*voice: Union[InputFile, str], business_connection_id: Optional[str] = None, message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendVoice*

Shortcut for method *aiogram.methods.send_voice.SendVoice* will automatically fill method attributes:

- chat_id
- reply_parameters

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as *aiogram.types.audio.Audio* or *aiogram.types.document.Document*). On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

Parameters

- **voice** – Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – Voice message caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the voice message caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the voice message in seconds
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_voice.SendVoice`

answer_paid_media(*star_count: int*, *media: List[InputPaidMediaPhoto | InputPaidMediaVideo]*, *business_connection_id: str | None = None*, *payload: str | None = None*, *caption: str | None = None*, *parse_mode: str | None = None*, *caption_entities: List[MessageEntity] | None = None*, *show_caption_above_media: bool | None = None*, *disable_notification: bool | None = None*, *protect_content: bool | None = None*, *reply_parameters: ReplyParameters | None = None*, *reply_markup: InlineKeyboardMarkup | ReplyKeyboardMarkup | ReplyKeyboardRemove | ForceReply | None = None*, ***kwargs: Any*) → *SendPaidMedia*

Shortcut for method `aiogram.methods.send_paid_media.SendPaidMedia` will automatically fill method attributes:

- `chat_id`

Use this method to send paid media. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpaidmedia

Parameters

- **star_count** – The number of Telegram Stars that must be paid to buy access to the media; 1-2500
- **media** – A JSON-serialized array describing the media to be sent; up to 10 items
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **payload** – Bot-defined paid media payload, 0-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **caption** – Media caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the media caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

Returns
instance of method `aiogram.methods.send_paid_media.SendPaidMedia`

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

reply_paid_media(*star_count: int*, *media: List[InputPaidMediaPhoto | InputPaidMediaVideo]*, *business_connection_id: str | None = None*, *payload: str | None = None*, *caption: str | None = None*, *parse_mode: str | None = None*, *caption_entities: List[MessageEntity] | None = None*, *show_caption_above_media: bool | None = None*, *disable_notification: bool | None = None*, *protect_content: bool | None = None*, *reply_markup: InlineKeyboardMarkup | ReplyKeyboardMarkup | ReplyKeyboardRemove | ForceReply | None = None*, ***kwargs: Any*) → *SendPaidMedia*

Shortcut for method `aiogram.methods.send_paid_media.SendPaidMedia` will automatically fill method attributes:

- `chat_id`
- `reply_parameters`

Use this method to send paid media. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpaidmedia

Parameters

- **star_count** – The number of Telegram Stars that must be paid to buy access to the media; 1-2500
- **media** – A JSON-serialized array describing the media to be sent; up to 10 items
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be sent
- **payload** – Bot-defined paid media payload, 0-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **caption** – Media caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the media caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

Returns

instance of method `aiogram.methods.send_paid_media.SendPaidMedia`

as_reply_parameters(*allow_sending_without_reply: Optional[Union[bool, Default]] = <Default('allow_sending_without_reply')>*, *quote: Optional[str] = None*, *quote_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>*, *quote_entities: Optional[List[MessageEntity]] = None*, *quote_position: Optional[int] = None*) → *ReplyParameters*

InlineKeyboardButton

class aiogram.types.inline_keyboard_button.**InlineKeyboardButton**(**, text: str, url: str | None = None, callback_data: str | None = None, web_app: WebAppInfo | None = None, login_url: LoginUrl | None = None, switch_inline_query: str | None = None, switch_inline_query_current_chat: str | None = None, switch_inline_query_chosen_chat: SwitchInlineQueryChosenChat | None = None, callback_game: CallbackGame | None = None, pay: bool | None = None, **extra_data: Any*)

This object represents one button of an inline keyboard. Exactly one of the optional fields must be used to specify type of the button.

Source: https://core.telegram.org/bots/api#inlinekeyboardbutton

text: str

Label text on the button

url: str | None

Optional. HTTP or tg:// URL to be opened when the button is pressed. Links `tg://user?id=<user_id>` can be used to mention a user by their identifier without using a username, if this is allowed by their privacy settings.

callback_data: str | None

Optional. Data to be sent in a callback query to the bot when the button is pressed, 1-64 bytes

web_app: *WebAppInfo* | None

Optional. Description of the Web App that will be launched when the user presses the button. The Web App will be able to send an arbitrary message on behalf of the user using the method *aiogram.methods.answer_web_app_query.AnswerWebAppQuery*. Available only in private chats between a user and the bot. Not supported for messages sent on behalf of a Telegram Business account.

login_url: *LoginUrl* | None

Optional. An HTTPS URL used to automatically authorize the user. Can be used as a replacement for the Telegram Login Widget.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

switch_inline_query: str | None

Optional. If set, pressing the button will prompt the user to select one of their chats, open that chat and insert the bot's username and the specified inline query in the input field. May be empty, in which case just the bot's username will be inserted. Not supported for messages sent on behalf of a Telegram Business account.

switch_inline_query_current_chat: str | None

Optional. If set, pressing the button will insert the bot's username and the specified inline query in the current chat's input field. May be empty, in which case only the bot's username will be inserted.

switch_inline_query_chosen_chat: *SwitchInlineQueryChosenChat* | None

Optional. If set, pressing the button will prompt the user to select one of their chats of the specified type, open that chat and insert the bot's username and the specified inline query in the input field. Not supported for messages sent on behalf of a Telegram Business account.

callback_game: *CallbackGame* | None

Optional. Description of the game that will be launched when the user presses the button.

pay: bool | None

Optional. Specify `True`, to send a Pay button. Substrings '' and 'XTR' in the buttons's text will be replaced with a Telegram Star icon.

InlineKeyboardMarkup

class aiogram.types.inline_keyboard_markup.**InlineKeyboardMarkup**(*, *inline_keyboard: List[List[*InlineKeyboardButton*]]*, ***extra_data: Any*)

This object represents an inline keyboard that appears right next to the message it belongs to.

Source: https://core.telegram.org/bots/api#inlinekeyboardmarkup

inline_keyboard: List[List[*InlineKeyboardButton*]]

Array of button rows, each represented by an Array of *aiogram.types.inline_keyboard_button.InlineKeyboardButton* objects

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

InputFile

class aiogram.types.input_file.**InputFile**(*filename: str | None = None*, *chunk_size: int = 65536*)

This object represents the contents of a file to be uploaded. Must be posted using multipart/form-data in the usual way that files are uploaded via the browser.

Source: https://core.telegram.org/bots/api#inputfile

abstract async read(*bot: Bot*) → AsyncGenerator[bytes, None]

class aiogram.types.input_file.**BufferedInputFile**(*file: bytes*, *filename: str*, *chunk_size: int = 65536*)

classmethod from_file(*path: str | Path*, *filename: str | None = None*, *chunk_size: int = 65536*) → *BufferedInputFile*

Create buffer from file

Parameters

- **path** – Path to file
- **filename** – Filename to be propagated to telegram. By default, will be parsed from path

- **`chunk_size`** – Uploading chunk size

Returns

instance of *`BufferedInputFile`*

`async` **`read`**(*bot: Bot*) → AsyncGenerator[bytes, None]

`class aiogram.types.input_file.`**`FSInputFile`**(*path: str | Path*, *filename: str | None = None*, *chunk_size: int = 65536*)

`async` **`read`**(*bot: Bot*) → AsyncGenerator[bytes, None]

`class aiogram.types.input_file.`**`URLInputFile`**(*url: str*, *headers: Dict[str, Any] | None = None*, *filename: str | None = None*, *chunk_size: int = 65536*, *timeout: int = 30*, *bot: 'Bot' | None = None*)

`async` **`read`**(*bot: Bot*) → AsyncGenerator[bytes, None]

InputMedia

`class aiogram.types.input_media.`**`InputMedia`**(***extra_data: Any*)

This object represents the content of a media message to be sent. It should be one of

- *`aiogram.types.input_media_animation.InputMediaAnimation`*
- *`aiogram.types.input_media_document.InputMediaDocument`*
- *`aiogram.types.input_media_audio.InputMediaAudio`*
- *`aiogram.types.input_media_photo.InputMediaPhoto`*
- *`aiogram.types.input_media_video.InputMediaVideo`*

Source: https://core.telegram.org/bots/api#inputmedia

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

InputMediaAnimation

class aiogram.types.input_media_animation.**InputMediaAnimation**(**, type: ~typing.Literal[InputMediaType.ANIMATION] = InputMediaType.ANIMATION, media: str | ~aiogram.types.input_file.InputFile, thumbnail: ~aiogram.types.input_file.InputFile | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntit | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, width: int | None = None, height: int | None = None, duration: int | None = None, has_spoiler: bool | None = None, **extra_data: ~typing.Any*)

Represents an animation file (GIF or H.264/MPEG-4 AVC video without sound) to be sent.

Source: https://core.telegram.org/bots/api#inputmediaanimation

type: Literal[InputMediaType.ANIMATION]

Type of the result, must be *animation*

media: str | ***InputFile***

File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*

thumbnail: ***InputFile*** **| None**

Optional. Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

caption: str | None

Optional. Caption of the animation to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the animation caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

width: int | None

Optional. Animation width

height: int | None

Optional. Animation height

duration: int | None

Optional. Animation duration in seconds

has_spoiler: bool | None

Optional. Pass `True` if the animation needs to be covered with a spoiler animation

InputMediaAudio

class aiogram.types.input_media_audio.**InputMediaAudio**(*, *type: ~typing.Literal[InputMediaType.AUDIO] = InputMediaType.AUDIO*, *media: str | ~aiogram.types.input_file.InputFile*, *thumbnail: ~aiogram.types.input_file.InputFile | None = None*, *caption: str | None = None*, *parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>*, *caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *duration: int | None = None*, *performer: str | None = None*, *title: str | None = None*, ***extra_data: ~typing.Any*)

Represents an audio file to be treated as music to be sent.

Source: https://core.telegram.org/bots/api#inputmediaaudio

type: Literal[InputMediaType.AUDIO]

Type of the result, must be *audio*

media: str | ***InputFile***

File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*

thumbnail: ***InputFile*** **| None**

Optional. Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

caption: str | None

Optional. Caption of the audio to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the audio caption. See formatting options for more details.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

duration: int | None

Optional. Duration of the audio in seconds

performer: str | None

Optional. Performer of the audio

title: str | None

Optional. Title of the audio

InputMediaDocument

class aiogram.types.input_media_document.**InputMediaDocument**(**, type: ~typing.Literal[InputMediaType.DOCUMENT] = InputMediaType.DOCUMENT, media: str | ~aiogram.types.input_file.InputFile, thumbnail: ~aiogram.types.input_file.InputFile | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, disable_content_type_detection: bool | None = None, **extra_data: ~typing.Any*)

Represents a general file to be sent.

Source: https://core.telegram.org/bots/api#inputmediadocument

type: Literal[InputMediaType.DOCUMENT]

Type of the result, must be *document*

media: str | *InputFile*

File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*

thumbnail: *InputFile* | None

Optional. Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

caption: str | None

Optional. Caption of the document to be sent, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the document caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

disable_content_type_detection: bool | None

Optional. Disables automatic server-side content type detection for files uploaded using multipart/form-data. Always `True`, if the document is sent as part of an album.

InputMediaPhoto

class aiogram.types.input_media_photo.**InputMediaPhoto**(**, type: ~typing.Literal[InputMediaType.PHOTO] = InputMediaType.PHOTO, media: str | ~aiogram.types.input_file.InputFile, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, has_spoiler: bool | None = None, **extra_data: ~typing.Any*)

Represents a photo to be sent.

Source: https://core.telegram.org/bots/api#inputmediaphoto

type: Literal[InputMediaType.PHOTO]

Type of the result, must be *photo*

media: str | *InputFile*

File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*

caption: str | None

Optional. Caption of the photo to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the photo caption. See formatting options for more details.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

has_spoiler: bool | None

Optional. Pass `True` if the photo needs to be covered with a spoiler animation

InputMediaVideo

class aiogram.types.input_media_video.**InputMediaVideo**(*, *type: ~typing.Literal[InputMediaType.VIDEO] = InputMediaType.VIDEO*, *media: str | ~aiogram.types.input_file.InputFile*, *thumbnail: ~aiogram.types.input_file.InputFile | None = None*, *caption: str | None = None*, *parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>*, *caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>*, *width: int | None = None*, *height: int | None = None*, *duration: int | None = None*, *supports_streaming: bool | None = None*, *has_spoiler: bool | None = None*, ***extra_data: ~typing.Any*)

Represents a video to be sent.

Source: https://core.telegram.org/bots/api#inputmediavideo

type: Literal[InputMediaType.VIDEO]

Type of the result, must be *video*

media: str | *InputFile*

File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*

thumbnail: *InputFile* | None

Optional. Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

caption: str | None

Optional. Caption of the video to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the video caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

width: int | None

Optional. Video width

height: int | None

Optional. Video height

duration: int | None

Optional. Video duration in seconds

supports_streaming: bool | None

Optional. Pass `True` if the uploaded video is suitable for streaming

has_spoiler: bool | None

Optional. Pass `True` if the video needs to be covered with a spoiler animation

InputPaidMedia

class aiogram.types.input_paid_media.**InputPaidMedia**(***extra_data: Any*)

This object describes the paid media to be sent. Currently, it can be one of

- *aiogram.types.input_paid_media_photo.InputPaidMediaPhoto*
- *aiogram.types.input_paid_media_video.InputPaidMediaVideo*

Source: https://core.telegram.org/bots/api#inputpaidmedia

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

InputPaidMediaPhoto

class aiogram.types.input_paid_media_photo.**InputPaidMediaPhoto**(*, *type: Literal[InputPaidMediaType.PHOTO] = InputPaidMediaType.PHOTO*, *media: str* | InputFile, ***extra_data: Any*)

The paid media to send is a photo.

Source: https://core.telegram.org/bots/api#inputpaidmediaphoto

type: Literal[InputPaidMediaType.PHOTO]

Type of the media, must be *photo*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

media: str | ***InputFile***

File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*

InputPaidMediaVideo

class aiogram.types.input_paid_media_video.**InputPaidMediaVideo**(*, *type: Literal[InputPaidMediaType.VIDEO] = InputPaidMediaType.VIDEO*, *media: str* | InputFile, *thumbnail:* InputFile | *str* | *None = None*, *width: int* | *None = None*, *height: int* | *None = None*, *duration: int* | *None = None*, *supports_streaming: bool* | *None = None*, ***extra_data: Any*)

The paid media to send is a video.

Source: https://core.telegram.org/bots/api#inputpaidmediavideo

type: Literal[InputPaidMediaType.VIDEO]

Type of the media, must be *video*

media: str | ***InputFile***

File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*

thumbnail: ***InputFile*** **| str | None**

Optional. Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

width: int | None

Optional. Video width

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

height: int | None

Optional. Video height

duration: int | None

Optional. Video duration in seconds

supports_streaming: bool | None

Optional. Pass `True` if the uploaded video is suitable for streaming

InputPollOption

class aiogram.types.input_poll_option.**InputPollOption**(**, text: str, text_parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, text_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, **extra_data: ~typing.Any*)

This object contains information about one answer option in a poll to be sent.

Source: https://core.telegram.org/bots/api#inputpolloption

text: str

Option text, 1-100 characters

text_parse_mode: str | Default | None

Optional. Mode for parsing entities in the text. See formatting options for more details. Currently, only custom emoji entities are allowed

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

text_entities: List[*MessageEntity*] | None

Optional. A JSON-serialized list of special entities that appear in the poll option text. It can be specified instead of *text_parse_mode*

KeyboardButton

class aiogram.types.keyboard_button.**KeyboardButton**(**, text: str, request_users: KeyboardButtonRequestUsers | None = None, request_chat: KeyboardButtonRequestChat | None = None, request_contact: bool | None = None, request_location: bool | None = None, request_poll: KeyboardButtonPollType | None = None, web_app: WebAppInfo | None = None, request_user: KeyboardButtonRequestUser | None = None, **extra_data: Any*)

This object represents one button of the reply keyboard. At most one of the optional fields must be used to specify type of the button. For simple text buttons, *String* can be used instead of this object to specify the button text. **Note:** *request_users* and *request_chat* options will only work in Telegram versions released after 3 February, 2023. Older clients will display *unsupported message*.

Source: https://core.telegram.org/bots/api#keyboardbutton

text: str

Text of the button. If none of the optional fields are used, it will be sent as a message when the button is pressed

request_users: *KeyboardButtonRequestUsers* | None

Optional. If specified, pressing the button will open a list of suitable users. Identifiers of selected users will be sent to the bot in a 'users_shared' service message. Available in private chats only.

request_chat: *KeyboardButtonRequestChat* | None

Optional. If specified, pressing the button will open a list of suitable chats. Tapping on a chat will send its identifier to the bot in a 'chat_shared' service message. Available in private chats only.

request_contact: bool | None

Optional. If True, the user's phone number will be sent as a contact when the button is pressed. Available in private chats only.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

request_location: bool | None

Optional. If True, the user's current location will be sent when the button is pressed. Available in private chats only.

request_poll: *KeyboardButtonPollType* | None

Optional. If specified, the user will be asked to create a poll and send it to the bot when the button is pressed. Available in private chats only.

web_app: *WebAppInfo* | None

Optional. If specified, the described Web App will be launched when the button is pressed. The Web App will be able to send a 'web_app_data' service message. Available in private chats only.

request_user: *KeyboardButtonRequestUser* | None

Optional. If specified, pressing the button will open a list of suitable users. Tapping on any user will send their identifier to the bot in a 'user_shared' service message. Available in private chats only.

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

KeyboardButtonPollType

class aiogram.types.keyboard_button_poll_type.**KeyboardButtonPollType**(**, type: str | None = None, **extra_data: Any*)

This object represents type of a poll, which is allowed to be created and sent when the corresponding button is pressed.

Source: https://core.telegram.org/bots/api#keyboardbuttonpolltype

type: str | None

Optional. If *quiz* is passed, the user will be allowed to create only polls in the quiz mode. If *regular* is passed, only regular polls will be allowed. Otherwise, the user will be allowed to create a poll of any type.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

KeyboardButtonRequestChat

class aiogram.types.keyboard_button_request_chat.**KeyboardButtonRequestChat**(**, request_id: int, chat_is_channel: bool, chat_is_forum: bool | None = None, chat_has_username: bool | None = None, chat_is_created: bool | None = None, user_administrator_rights: ChatAdministratorRights | None = None, bot_administrator_rights: ChatAdministratorRights | None = None, bot_is_member: bool | None = None, request_title: bool | None = None, request_username: bool | None = None, request_photo: bool | None = None, **extra_data: Any*)

This object defines the criteria used to request a suitable chat. Information about the selected chat will be shared with the bot when the corresponding button is pressed. The bot will be granted requested rights in the chat if appropriate. More about requesting chats ».

Source: https://core.telegram.org/bots/api#keyboardbuttonrequestchat

request_id: int

Signed 32-bit identifier of the request, which will be received back in the *aiogram.types.chat_shared.ChatShared* object. Must be unique within the message

chat_is_channel: bool

Pass `True` to request a channel chat, pass `False` to request a group or a supergroup chat.

chat_is_forum: bool | None

Optional. Pass `True` to request a forum supergroup, pass `False` to request a non-forum chat. If not specified, no additional restrictions are applied.

chat_has_username: bool | None

Optional. Pass `True` to request a supergroup or a channel with a username, pass `False` to request a chat without a username. If not specified, no additional restrictions are applied.

chat_is_created: bool | None

Optional. Pass `True` to request a chat owned by the user. Otherwise, no additional restrictions are applied.

user_administrator_rights: *ChatAdministratorRights* | None

Optional. A JSON-serialized object listing the required administrator rights of the user in the chat. The rights must be a superset of *bot_administrator_rights*. If not specified, no additional restrictions are applied.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

bot_administrator_rights: *ChatAdministratorRights* | None

Optional. A JSON-serialized object listing the required administrator rights of the bot in the chat. The rights must be a subset of *user_administrator_rights*. If not specified, no additional restrictions are applied.

bot_is_member: bool | None

Optional. Pass `True` to request a chat with the bot as a member. Otherwise, no additional restrictions are applied.

request_title: bool | None

Optional. Pass `True` to request the chat's title

request_username: bool | None

Optional. Pass `True` to request the chat's username

request_photo: bool | None

Optional. Pass `True` to request the chat's photo

KeyboardButtonRequestUser

class aiogram.types.keyboard_button_request_user.**KeyboardButtonRequestUser**(**, request_id: int, user_is_bot: bool | None = None, user_is_premium: bool | None = None, **extra_data: Any*)

This object defines the criteria used to request a suitable user. The identifier of the selected user will be shared with the bot when the corresponding button is pressed. More about requesting users »

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Source: https://core.telegram.org/bots/api#keyboardbuttonrequestuser

request_id: int

Signed 32-bit identifier of the request, which will be received back in the `aiogram.types.user_shared.UserShared` object. Must be unique within the message

user_is_bot: bool | None

Optional. Pass `True` to request a bot, pass `False` to request a regular user. If not specified, no additional restrictions are applied.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_is_premium: bool | None

Optional. Pass `True` to request a premium user, pass `False` to request a non-premium user. If not specified, no additional restrictions are applied.

KeyboardButtonRequestUsers

class aiogram.types.keyboard_button_request_users.**KeyboardButtonRequestUsers**(**, request_id: int, user_is_bot: bool | None = None, user_is_premium: bool | None = None, max_quantity: int | None = None, request_name: bool | None = None, request_username: bool | None = None, request_photo: bool | None = None, **extra_data: Any*)

This object defines the criteria used to request suitable users. Information about the selected users will be shared with the bot when the corresponding button is pressed. More about requesting users »

Source: https://core.telegram.org/bots/api#keyboardbuttonrequestusers

request_id: int

Signed 32-bit identifier of the request that will be received back in the `aiogram.types.users_shared.UsersShared` object. Must be unique within the message

user_is_bot: bool | None

Optional. Pass `True` to request bots, pass `False` to request regular users. If not specified, no additional restrictions are applied.

user_is_premium: bool | None

Optional. Pass `True` to request premium users, pass `False` to request non-premium users. If not specified, no additional restrictions are applied.

max_quantity: int | None

Optional. The maximum number of users to be selected; 1-10. Defaults to 1.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

request_name: bool | None

Optional. Pass `True` to request the users' first and last names

request_username: bool | None

Optional. Pass `True` to request the users' usernames

request_photo: bool | None

Optional. Pass `True` to request the users' photos

LinkPreviewOptions

class aiogram.types.link_preview_options.**LinkPreviewOptions**(**, is_disabled: bool | ~aiogram.client.default.Default | None = <Default('link_preview_is_disabled')>, url: str | None = None, prefer_small_media: bool | ~aiogram.client.default.Default | None = <Default('link_preview_prefer_small_media')>, prefer_large_media: bool | ~aiogram.client.default.Default | None = <Default('link_preview_prefer_large_media')>, show_above_text: bool | ~aiogram.client.default.Default | None = <Default('link_preview_show_above_text')>, **extra_data: ~typing.Any*)

Describes the options used for link preview generation.

Source: https://core.telegram.org/bots/api#linkpreviewoptions

is_disabled: bool | Default | None

Optional. `True`, if the link preview is disabled

url: str | None

Optional. URL to use for the link preview. If empty, then the first URL found in the message text will be used

prefer_small_media: bool | Default | None

Optional. `True`, if the media in the link preview is supposed to be shrunk; ignored if the URL isn't explicitly specified or media size change isn't supported for the preview

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

prefer_large_media: bool | Default | None

Optional. `True`, if the media in the link preview is supposed to be enlarged; ignored if the URL isn't explicitly specified or media size change isn't supported for the preview

show_above_text: bool | Default | None

Optional. `True`, if the link preview must be shown above the message text; otherwise, the link preview will be shown below the message text

Location

class `aiogram.types.location.`**`Location`**(**, latitude: float, longitude: float, horizontal_accuracy: float | None = None, live_period: int | None = None, heading: int | None = None, proximity_alert_radius: int | None = None, **extra_data: Any*)

This object represents a point on the map.

Source: https://core.telegram.org/bots/api#location

`latitude: float`

Latitude as defined by the sender

`longitude: float`

Longitude as defined by the sender

`horizontal_accuracy: float | None`

Optional. The radius of uncertainty for the location, measured in meters; 0-1500

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`live_period: int | None`

Optional. Time relative to the message sending date, during which the location can be updated; in seconds. For active live locations only.

`heading: int | None`

Optional. The direction in which user is moving, in degrees; 1-360. For active live locations only.

`proximity_alert_radius: int | None`

Optional. The maximum distance for proximity alerts about approaching another chat member, in meters. For sent live locations only.

LoginUrl

class `aiogram.types.login_url.`**`LoginUrl`**(**, url: str, forward_text: str | None = None, bot_username: str | None = None, request_write_access: bool | None = None, **extra_data: Any*)

This object represents a parameter of the inline keyboard button used to automatically authorize a user. Serves as a great replacement for the Telegram Login Widget when the user is coming from Telegram. All the user needs to do is tap/click a button and confirm that they want to log in: Telegram apps support these buttons as of version 5.7.

Sample bot: @discussbot

Source: https://core.telegram.org/bots/api#loginurl

`url: str`

An HTTPS URL to be opened with user authorization data added to the query string when the button is pressed. If the user refuses to provide authorization data, the original URL without information about the user will be opened. The data added is the same as described in Receiving authorization data.

forward_text: str | None

Optional. New text of the button in forwarded messages.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

bot_username: str | None

Optional. Username of a bot, which will be used for user authorization. See Setting up a bot for more details. If not specified, the current bot's username will be assumed. The *url*'s domain must be the same as the domain linked with the bot. See Linking your domain to the bot for more details.

request_write_access: bool | None

Optional. Pass `True` to request the permission for your bot to send messages to the user.

MaybeInaccessibleMessage

class aiogram.types.maybe_inaccessible_message.**MaybeInaccessibleMessage**(***extra_data: Any*)

This object describes a message that can be inaccessible to the bot. It can be one of

- *aiogram.types.message.Message*
- *aiogram.types.inaccessible_message.InaccessibleMessage*

Source: https://core.telegram.org/bots/api#maybeinaccessiblemessage

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

MenuButton

class aiogram.types.menu_button.**MenuButton**(**, type: str, text: str | None = None, web_app:* WebAppInfo *| None = None, **extra_data: Any*)

This object describes the bot's menu button in a private chat. It should be one of

- *aiogram.types.menu_button_commands.MenuButtonCommands*
- *aiogram.types.menu_button_web_app.MenuButtonWebApp*
- *aiogram.types.menu_button_default.MenuButtonDefault*

If a menu button other than *aiogram.types.menu_button_default.MenuButtonDefault* is set for a private chat, then it is applied in the chat. Otherwise the default menu button is applied. By default, the menu button opens the list of bot commands.

Source: https://core.telegram.org/bots/api#menubutton

type: str

Type of the button

text: str | None

Optional. Text on the button

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

web_app: *WebAppInfo* | None

Optional. Description of the Web App that will be launched when the user presses the button. The Web App will be able to send an arbitrary message on behalf of the user using the method *aiogram.methods.answer_web_app_query.AnswerWebAppQuery*. Alternatively, a `t.me` link to a Web App of the bot can be specified in the object instead of the Web App's URL, in which case the Web App will be opened as if the user pressed the link.

MenuButtonCommands

class aiogram.types.menu_button_commands.**MenuButtonCommands**(*, *type: Literal[MenuButtonType.COMMANDS] = MenuButtonType.COMMANDS*, *text: str | None = None*, *web_app: WebAppInfo | None = None*, ***extra_data: Any*)

Represents a menu button, which opens the bot's list of commands.

Source: https://core.telegram.org/bots/api#menubuttoncommands

type: Literal[MenuButtonType.COMMANDS]

Type of the button, must be *commands*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

MenuButtonDefault

class aiogram.types.menu_button_default.**MenuButtonDefault**(*, *type: Literal[MenuButtonType.DEFAULT] = MenuButtonType.DEFAULT*, *text: str | None = None*, *web_app: WebAppInfo | None = None*, ***extra_data: Any*)

Describes that no specific value for the menu button was set.

Source: https://core.telegram.org/bots/api#menubuttondefault

type: Literal[MenuButtonType.DEFAULT]

Type of the button, must be *default*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

MenuButtonWebApp

class `aiogram.types.menu_button_web_app.`**`MenuButtonWebApp`**(*, *type: Literal[MenuButtonType.WEB_APP] = MenuButtonType.WEB_APP*, *text: str*, *web_app: WebAppInfo*, ***extra_data: Any*)

Represents a menu button, which launches a Web App.

Source: https://core.telegram.org/bots/api#menubuttonwebapp

`type: Literal[MenuButtonType.WEB_APP]`

Type of the button, must be *web_app*

`text: str`

Text on the button

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`web_app:` *`WebAppInfo`*

Description of the Web App that will be launched when the user presses the button. The Web App will be able to send an arbitrary message on behalf of the user using the method *`aiogram.methods.answer_web_app_query.AnswerWebAppQuery`*. Alternatively, a `t.me` link to a Web App of the bot can be specified in the object instead of the Web App's URL, in which case the Web App will be opened as if the user pressed the link.

Message

class aiogram.types.message.**Message**(**, message_id: int, date: _datetime_serializer, return_type=int, when_used=unless - none)], chat: Chat, message_thread_id: int | None = None, from_user: User | None = None, sender_chat: Chat | None = None, sender_boost_count: int | None = None, sender_business_bot: User | None = None, business_connection_id: str | None = None, forward_origin: MessageOriginUser | MessageOriginHiddenUser | MessageOriginChat | MessageOriginChannel | None = None, is_topic_message: bool | None = None, is_automatic_forward: bool | None = None, reply_to_message: Message | None = None, external_reply: ExternalReplyInfo | None = None, quote: TextQuote | None = None, reply_to_story: Story | None = None, via_bot: User | None = None, edit_date: int | None = None, has_protected_content: bool | None = None, is_from_offline: bool | None = None, media_group_id: str | None = None, author_signature: str | None = None, text: str | None = None, entities: List[MessageEntity] | None = None, link_preview_options: LinkPreviewOptions | None = None, effect_id: str | None = None, animation: Animation | None = None, audio: Audio | None = None, document: Document | None = None, paid_media: PaidMediaInfo | None = None, photo: List[PhotoSize] | None = None, sticker: Sticker | None = None, story: Story | None = None, video: Video | None = None, video_note: VideoNote | None = None, voice: Voice | None = None, caption: str | None = None, caption_entities: List[MessageEntity] | None = None, show_caption_above_media: bool | None = None, has_media_spoiler: bool | None = None, contact: Contact | None = None, dice: Dice | None = None, game: Game | None = None, poll: Poll | None = None, venue: Venue | None = None, location: Location | None = None, new_chat_members: List[User] | None = None, left_chat_member: User | None = None, new_chat_title: str | None = None, new_chat_photo: List[PhotoSize] | None = None, delete_chat_photo: bool | None = None, group_chat_created: bool | None = None, supergroup_chat_created: bool | None = None, channel_chat_created: bool | None = None, message_auto_delete_timer_changed: MessageAutoDeleteTimerChanged | None = None, migrate_to_chat_id: int | None = None, migrate_from_chat_id: int | None = None, pinned_message: Message | InaccessibleMessage | None = None, invoice: Invoice | None = None, successful_payment: SuccessfulPayment | None = None, refunded_payment: RefundedPayment | None = None, users_shared: UsersShared | None = None, chat_shared: ChatShared | None = None, connected_website: str | None = None, write_access_allowed: WriteAccessAllowed | None = None, passport_data: PassportData | None = None, proximity_alert_triggered: ProximityAlertTriggered | None = None, boost_added: ChatBoostAdded | None = None, chat_background_set: ChatBackground | None = None, forum_topic_created: ForumTopicCreated | None = None, forum_topic_edited: ForumTopicEdited | None = None, forum_topic_closed: ForumTopicClosed | None = None, forum_topic_reopened: ForumTopicReopened | None = None, general_forum_topic_hidden: GeneralForumTopicHidden | None = None, general_forum_topic_unhidden: GeneralForumTopicUnhidden | None = None, giveaway_created: GiveawayCreated | None = None, giveaway: Giveaway | None = None, giveaway_winners: GiveawayWinners | None = None, giveaway_completed: GiveawayCompleted | None = None, video_chat_scheduled: VideoChatScheduled | None = None, video_chat_started: VideoChatStarted | None = None, video_chat_ended: VideoChatEnded | None = None, video_chat_participants_invited:*

This object represents a message.

Source: https://core.telegram.org/bots/api#message

message_id: int

Unique message identifier inside this chat

date: DateTime

Date the message was sent in Unix time. It is always a positive number, representing a valid date.

chat: *Chat*

Chat the message belongs to

message_thread_id: int | None

Optional. Unique identifier of a message thread to which the message belongs; for supergroups only

from_user: *User* | None

Optional. Sender of the message; may be empty for messages sent to channels. For backward compatibility, if the message was sent on behalf of a chat, the field contains a fake sender user in non-channel chats

sender_chat: *Chat* | None

Optional. Sender of the message when sent on behalf of a chat. For example, the supergroup itself for messages sent by its anonymous administrators or a linked channel for messages automatically forwarded to the channel's discussion group. For backward compatibility, if the message was sent on behalf of a chat, the field *from* contains a fake sender user in non-channel chats.

sender_boost_count: int | None

Optional. If the sender of the message boosted the chat, the number of boosts added by the user

sender_business_bot: *User* | None

Optional. The bot that actually sent the message on behalf of the business account. Available only for outgoing messages sent on behalf of the connected business account.

business_connection_id: str | None

Optional. Unique identifier of the business connection from which the message was received. If non-empty, the message belongs to a chat of the corresponding business account that is independent from any potential bot chat which might share the same identifier.

forward_origin: *MessageOriginUser* | *MessageOriginHiddenUser* | *MessageOriginChat* | *MessageOriginChannel* | None

Optional. Information about the original message for forwarded messages

is_topic_message: bool | None

Optional. `True`, if the message is sent to a forum topic

is_automatic_forward: bool | None

Optional. `True`, if the message is a channel post that was automatically forwarded to the connected discussion group

reply_to_message: *Message* | None

Optional. For replies in the same chat and message thread, the original message. Note that the Message object in this field will not contain further *reply_to_message* fields even if it itself is a reply.

external_reply: *ExternalReplyInfo* | None

Optional. Information about the message that is being replied to, which may come from another chat or forum topic

quote: *TextQuote* **| None**

Optional. For replies that quote part of the original message, the quoted part of the message

reply_to_story: *Story* **| None**

Optional. For replies to a story, the original story

via_bot: *User* **| None**

Optional. Bot through which the message was sent

edit_date: int | None

Optional. Date the message was last edited in Unix time

has_protected_content: bool | None

Optional. `True`, if the message can't be forwarded

is_from_offline: bool | None

Optional. True, if the message was sent by an implicit action, for example, as an away or a greeting business message, or as a scheduled message

media_group_id: str | None

Optional. The unique identifier of a media message group this message belongs to

author_signature: str | None

Optional. Signature of the post author for messages in channels, or the custom title of an anonymous group administrator

text: str | None

Optional. For text messages, the actual UTF-8 text of the message

entities: List[*MessageEntity*] **| None**

Optional. For text messages, special entities like usernames, URLs, bot commands, etc. that appear in the text

link_preview_options: *LinkPreviewOptions* **| None**

Optional. Options used for link preview generation for the message, if it is a text message and link preview options were changed

effect_id: str | None

Optional. Unique identifier of the message effect added to the message

animation: *Animation* **| None**

Optional. Message is an animation, information about the animation. For backward compatibility, when this field is set, the *document* field will also be set

audio: *Audio* **| None**

Optional. Message is an audio file, information about the file

document: *Document* **| None**

Optional. Message is a general file, information about the file

paid_media: *PaidMediaInfo* **| None**

Optional. Message contains paid media; information about the paid media

photo: List[*PhotoSize*] **| None**

Optional. Message is a photo, available sizes of the photo

sticker: *Sticker* **| None**

Optional. Message is a sticker, information about the sticker

story: *Story* | None

Optional. Message is a forwarded story

video: *Video* | None

Optional. Message is a video, information about the video

video_note: *VideoNote* | None

Optional. Message is a video note, information about the video message

voice: *Voice* | None

Optional. Message is a voice message, information about the file

caption: str | None

Optional. Caption for the animation, audio, document, paid media, photo, video or voice

caption_entities: List[*MessageEntity*] | None

Optional. For messages with a caption, special entities like usernames, URLs, bot commands, etc. that appear in the caption

show_caption_above_media: bool | None

Optional. True, if the caption must be shown above the message media

has_media_spoiler: bool | None

Optional. `True`, if the message media is covered by a spoiler animation

contact: *Contact* | None

Optional. Message is a shared contact, information about the contact

dice: *Dice* | None

Optional. Message is a dice with random value

game: *Game* | None

Optional. Message is a game, information about the game. More about games »

poll: *Poll* | None

Optional. Message is a native poll, information about the poll

venue: *Venue* | None

Optional. Message is a venue, information about the venue. For backward compatibility, when this field is set, the *location* field will also be set

location: *Location* | None

Optional. Message is a shared location, information about the location

new_chat_members: List[*User*] | None

Optional. New members that were added to the group or supergroup and information about them (the bot itself may be one of these members)

left_chat_member: *User* | None

Optional. A member was removed from the group, information about them (this member may be the bot itself)

new_chat_title: str | None

Optional. A chat title was changed to this value

new_chat_photo: List[*PhotoSize*] | None

Optional. A chat photo was change to this value

delete_chat_photo: bool | None

Optional. Service message: the chat photo was deleted

group_chat_created: bool | None

Optional. Service message: the group has been created

supergroup_chat_created: bool | None

Optional. Service message: the supergroup has been created. This field can't be received in a message coming through updates, because bot can't be a member of a supergroup when it is created. It can only be found in reply_to_message if someone replies to a very first message in a directly created supergroup.

channel_chat_created: bool | None

Optional. Service message: the channel has been created. This field can't be received in a message coming through updates, because bot can't be a member of a channel when it is created. It can only be found in reply_to_message if someone replies to a very first message in a channel.

message_auto_delete_timer_changed: *MessageAutoDeleteTimerChanged* | None

Optional. Service message: auto-delete timer settings changed in the chat

migrate_to_chat_id: int | None

Optional. The group has been migrated to a supergroup with the specified identifier. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this identifier.

migrate_from_chat_id: int | None

Optional. The supergroup has been migrated from a group with the specified identifier. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this identifier.

pinned_message: *Message* | *InaccessibleMessage* | None

Optional. Specified message was pinned. Note that the Message object in this field will not contain further *reply_to_message* fields even if it itself is a reply.

invoice: *Invoice* | None

Optional. Message is an invoice for a payment, information about the invoice. More about payments »

successful_payment: *SuccessfulPayment* | None

Optional. Message is a service message about a successful payment, information about the payment. More about payments »

refunded_payment: *RefundedPayment* | None

Optional. Message is a service message about a refunded payment, information about the payment. More about payments »

users_shared: *UsersShared* | None

Optional. Service message: users were shared with the bot

chat_shared: *ChatShared* | None

Optional. Service message: a chat was shared with the bot

connected_website: str | None

Optional. The domain name of the website on which the user has logged in. More about Telegram Login »

write_access_allowed: *WriteAccessAllowed* **| None**

Optional. Service message: the user allowed the bot to write messages after adding it to the attachment or side menu, launching a Web App from a link, or accepting an explicit request from a Web App sent by the method requestWriteAccess

passport_data: *PassportData* **| None**

Optional. Telegram Passport data

proximity_alert_triggered: *ProximityAlertTriggered* **| None**

Optional. Service message. A user in the chat triggered another user's proximity alert while sharing Live Location.

boost_added: *ChatBoostAdded* **| None**

Optional. Service message: user boosted the chat

chat_background_set: *ChatBackground* **| None**

Optional. Service message: chat background set

forum_topic_created: *ForumTopicCreated* **| None**

Optional. Service message: forum topic created

forum_topic_edited: *ForumTopicEdited* **| None**

Optional. Service message: forum topic edited

forum_topic_closed: *ForumTopicClosed* **| None**

Optional. Service message: forum topic closed

forum_topic_reopened: *ForumTopicReopened* **| None**

Optional. Service message: forum topic reopened

general_forum_topic_hidden: *GeneralForumTopicHidden* **| None**

Optional. Service message: the 'General' forum topic hidden

model_computed_fields: **ClassVar[Dict[str, ComputedFieldInfo]] = {}**

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

general_forum_topic_unhidden: *GeneralForumTopicUnhidden* **| None**

Optional. Service message: the 'General' forum topic unhidden

giveaway_created: *GiveawayCreated* **| None**

Optional. Service message: a scheduled giveaway was created

giveaway: *Giveaway* **| None**

Optional. The message is a scheduled giveaway message

giveaway_winners: *GiveawayWinners* **| None**

Optional. A giveaway with public winners was completed

giveaway_completed: *GiveawayCompleted* **| None**

Optional. Service message: a giveaway without public winners was completed

video_chat_scheduled: *VideoChatScheduled* **| None**

Optional. Service message: video chat scheduled

video_chat_started: ***VideoChatStarted*** **| None**

Optional. Service message: video chat started

video_chat_ended: ***VideoChatEnded*** **| None**

Optional. Service message: video chat ended

video_chat_participants_invited: ***VideoChatParticipantsInvited*** **| None**

Optional. Service message: new participants invited to a video chat

web_app_data: ***WebAppData*** **| None**

Optional. Service message: data sent by a Web App

reply_markup: ***InlineKeyboardMarkup*** **| None**

Optional. Inline keyboard attached to the message. `login_url` buttons are represented as ordinary `url` buttons.

forward_date: DateTime | None

Optional. For forwarded messages, date the original message was sent in Unix time

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

forward_from: ***User*** **| None**

Optional. For forwarded messages, sender of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

forward_from_chat: ***Chat*** **| None**

Optional. For messages forwarded from channels or from anonymous administrators, information about the original sender chat

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

forward_from_message_id: int | None

Optional. For messages forwarded from channels, identifier of the original message in the channel

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

forward_sender_name: str | None

Optional. Sender's name for messages forwarded from users who disallow adding a link to their account in forwarded messages

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

forward_signature: str | None

Optional. For forwarded messages that were originally sent in channels or by an anonymous chat administrator, signature of the message sender if present

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

user_shared: ***UserShared*** **| None**

Optional. Service message: a user was shared with the bot

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

property content_type: str

property html_text: str

property md_text: str

as_reply_parameters(*allow_sending_without_reply: Optional[Union[bool, Default]] = <Default('allow_sending_without_reply')>, quote: Optional[str] = None, quote_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, quote_entities: Optional[List[MessageEntity]] = None, quote_position: Optional[int] = None*) → *ReplyParameters*

reply_animation(*animation: Union[InputFile, str], duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendAnimation*

Shortcut for method `aiogram.methods.send_animation.SendAnimation` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

Parameters

- **animation** – Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*
- **duration** – Duration of sent animation in seconds
- **width** – Animation width
- **height** – Animation height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the animation caption. See formatting options for more details.

- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the animation needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method `aiogram.methods.send_animation.SendAnimation`

answer_animation(*animation: Union[InputFile, str], duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAnimation*

Shortcut for method `aiogram.methods.send_animation.SendAnimation` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

Parameters

- **animation** – Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*
- **duration** – Duration of sent animation in seconds

- **width** – Animation width
- **height** – Animation height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the animation caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the animation needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_animation.SendAnimation`

reply_audio(*audio: Union[InputFile, str], caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, performer: Optional[str] = None, title: Optional[str] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendAudio*

Shortcut for method `aiogram.methods.send_audio.SendAudio` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`

- `business_connection_id`
- `reply_parameters`

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent *aiogram.types.message.Message* is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the *aiogram.methods.send_voice.SendVoice* method instead.

Source: https://core.telegram.org/bots/api#sendaudio

Parameters

- **audio** – Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **caption** – Audio caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the audio caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method *aiogram.methods.send_audio.SendAudio*

answer_audio(*audio: Union[InputFile, str], caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, performer: Optional[str] = None, title: Optional[str] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendAudio*

Shortcut for method `aiogram.methods.send_audio.SendAudio` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the `aiogram.methods.send_voice.SendVoice` method instead.

Source: https://core.telegram.org/bots/api#sendaudio

Parameters

- **audio** – Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **caption** – Audio caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the audio caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only

- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_audio.SendAudio*

reply_contact(*phone_number: str*, *first_name: str*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → *SendContact*

Shortcut for method *aiogram.methods.send_contact.SendContact* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send phone contacts. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendcontact

Parameters

- **phone_number** – Contact's phone number
- **first_name** – Contact's first name
- **last_name** – Contact's last name
- **vcard** – Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method *aiogram.methods.send_contact.SendContact*

answer_contact(*phone_number: str, first_name: str, last_name: Optional[str] = None, vcard: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendContact*

Shortcut for method `aiogram.methods.send_contact.SendContact` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send phone contacts. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendcontact

Parameters

- **phone_number** – Contact's phone number
- **first_name** – Contact's first name
- **last_name** – Contact's last name
- **vcard** – Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_contact.SendContact`

reply_document(*document: Union[InputFile, str], thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, disable_content_type_detection: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendDocument*

Shortcut for method `aiogram.methods.send_document.SendDocument` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send general files. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

Parameters

- **document** – File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the document caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method `aiogram.methods.send_document.SendDocument`

answer_document(*document: Union[InputFile, str], thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, disable_content_type_detection: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendDocument*

Shortcut for method `aiogram.methods.send_document.SendDocument` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send general files. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

Parameters

- **document** – File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the document caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to

- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_document.SendDocument`

reply_game(*game_short_name: str*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → SendGame

Shortcut for method `aiogram.methods.send_game.SendGame` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send a game. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendgame

Parameters

- **game_short_name** – Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_game.SendGame`

answer_game(*game_short_name: str*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → SendGame

Shortcut for method `aiogram.methods.send_game.SendGame` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send a game. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendgame

Parameters

- **game_short_name** – Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_game.SendGame*

reply_invoice(*title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], provider_token: Optional[str] = None, max_tip_amount: Optional[int] = None, suggested_tip_amounts: Optional[List[int]] = None, start_parameter: Optional[str] = None, provider_data: Optional[str] = None, photo_url: Optional[str] = None, photo_size: Optional[int] = None, photo_width: Optional[int] = None, photo_height: Optional[int] = None, need_name: Optional[bool] = None, need_phone_number: Optional[bool] = None, need_email: Optional[bool] = None, need_shipping_address: Optional[bool] = None, send_phone_number_to_provider: Optional[bool] = None, send_email_to_provider: Optional[bool] = None, is_flexible: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendInvoice*

Shortcut for method *aiogram.methods.send_invoice.SendInvoice* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send invoices. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters

- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **currency** – Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.
- **prices** – Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.
- **provider_token** – Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.
- **max_tip_amount** – The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.
- **start_parameter** – Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter
- **provider_data** – JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.
- **need_phone_number** – Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.
- **need_email** – Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.
- **need_shipping_address** – Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.
- **send_phone_number_to_provider** – Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.
- **send_email_to_provider** – Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.
- **is_flexible** – Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.

- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method `aiogram.methods.send_invoice.SendInvoice`

answer_invoice(*title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], provider_token: Optional[str] = None, max_tip_amount: Optional[int] = None, suggested_tip_amounts: Optional[List[int]] = None, start_parameter: Optional[str] = None, provider_data: Optional[str] = None, photo_url: Optional[str] = None, photo_size: Optional[int] = None, photo_width: Optional[int] = None, photo_height: Optional[int] = None, need_name: Optional[bool] = None, need_phone_number: Optional[bool] = None, need_email: Optional[bool] = None, need_shipping_address: Optional[bool] = None, send_phone_number_to_provider: Optional[bool] = None, send_email_to_provider: Optional[bool] = None, is_flexible: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[InlineKeyboardMarkup] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendInvoice*

Shortcut for method `aiogram.methods.send_invoice.SendInvoice` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send invoices. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **currency** – Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.
- **prices** – Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.
- **provider_token** – Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.

- **max_tip_amount** – The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.
- **start_parameter** – Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter
- **provider_data** – JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.
- **need_phone_number** – Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.
- **need_email** – Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.
- **need_shipping_address** – Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.
- **send_phone_number_to_provider** – Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.
- **send_email_to_provider** – Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.
- **is_flexible** – Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_invoice.SendInvoice*

reply_location(*latitude: float*, *longitude: float*, *horizontal_accuracy: Optional[float] = None*, *live_period: Optional[int] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → *SendLocation*

Shortcut for method *aiogram.methods.send_location.SendLocation* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send point on the map. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendlocation

Parameters

- **latitude** – Latitude of the location
- **longitude** – Longitude of the location
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **live_period** – Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.
- **heading** – For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

- **allow_sending_without_reply** – Pass True if the message should be sent even if the specified replied-to message is not found

Returns
instance of method aiogram.methods.send_location.SendLocation

answer_location(*latitude: float*, *longitude: float*, *horizontal_accuracy: Optional[float] = None*, *live_period: Optional[int] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → *SendLocation*

Shortcut for method aiogram.methods.send_location.SendLocation will automatically fill method attributes:

- chat_id
- message_thread_id
- business_connection_id

Use this method to send point on the map. On success, the sent aiogram.types.message.Message is returned.

Source: https://core.telegram.org/bots/api#sendlocation

Parameters

- **latitude** – Latitude of the location
- **longitude** – Longitude of the location
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **live_period** – Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.
- **heading** – For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method *aiogram.methods.send_location.SendLocation*

reply_media_group(*media: List[Union[InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo]], disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendMediaGroup*

Shortcut for method *aiogram.methods.send_media_group.SendMediaGroup* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **media** – A JSON-serialized array describing messages to be sent, must include 2-10 items
- **disable_notification** – Sends messages silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent messages from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method *aiogram.methods.send_media_group.SendMediaGroup*

answer_media_group(*media: List[Union[InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo]], disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMediaGroup*

Shortcut for method *aiogram.methods.send_media_group.SendMediaGroup* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **media** – A JSON-serialized array describing messages to be sent, must include 2-10 items
- **disable_notification** – Sends messages silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent messages from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the messages are a reply, ID of the original message

Returns

instance of method *aiogram.methods.send_media_group.SendMediaGroup*

reply(*text: str, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, entities: Optional[List[MessageEntity]] = None, link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>, **kwargs: Any*) → *SendMessage*

Shortcut for method *aiogram.methods.send_message.SendMessage* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send text messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendmessage

Parameters

- **text** – Text of the message to be sent, 1-4096 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the message text. See formatting options for more details.
- **entities** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*
- **link_preview_options** – Link preview generation options for the message

- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **disable_web_page_preview** – Disables link previews for links in this message

Returns
instance of method `aiogram.methods.send_message.SendMessage`

answer(*text: str, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, entities: Optional[List[MessageEntity]] = None, link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendMessage*

Shortcut for method `aiogram.methods.send_message.SendMessage` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send text messages. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendmessage

Parameters

- **text** – Text of the message to be sent, 1-4096 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the message text. See formatting options for more details.
- **entities** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*
- **link_preview_options** – Link preview generation options for the message
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only

- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **disable_web_page_preview** – Disables link previews for links in this message
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_message.SendMessage`

reply_photo(*photo: Union[InputFile, str], caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendPhoto*

Shortcut for method `aiogram.methods.send_photo.SendPhoto` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send photos. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendphoto

Parameters

- **photo** – Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*
- **caption** – Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the photo caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the photo needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.

- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method *aiogram.methods.send_photo.SendPhoto*

answer_photo(*photo: Union[InputFile, str], caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPhoto*

Shortcut for method *aiogram.methods.send_photo.SendPhoto* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send photos. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendphoto

Parameters

- **photo** – Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*
- **caption** – Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the photo caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the photo needs to be covered with a spoiler animation
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.

- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_photo.SendPhoto`

reply_poll(*question: str, options: List[Union[InputPollOption, str]], question_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, question_entities: Optional[List[MessageEntity]] = None, is_anonymous: Optional[bool] = None, type: Optional[str] = None, allows_multiple_answers: Optional[bool] = None, correct_option_id: Optional[int] = None, explanation: Optional[str] = None, explanation_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, explanation_entities: Optional[List[MessageEntity]] = None, open_period: Optional[int] = None, close_date: Optional[Union[datetime.datetime, datetime.timedelta, int]] = None, is_closed: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendPoll*

Shortcut for method `aiogram.methods.send_poll.SendPoll` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send a native poll. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpoll

Parameters

- **question** – Poll question, 1-300 characters
- **options** – A JSON-serialized list of 2-10 answer options
- **question_parse_mode** – Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed
- **question_entities** – A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*
- **is_anonymous** – `True`, if the poll needs to be anonymous, defaults to `True`
- **type** – Poll type, 'quiz' or 'regular', defaults to 'regular'

- **allows_multiple_answers** – `True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`
- **correct_option_id** – 0-based identifier of the correct answer option, required for polls in quiz mode
- **explanation** – Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing
- **explanation_parse_mode** – Mode for parsing entities in the explanation. See formatting options for more details.
- **explanation_entities** – A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*
- **open_period** – Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.
- **close_date** – Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.
- **is_closed** – Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_poll.SendPoll`

answer_poll(*question: str, options: List[Union[InputPollOption, str]], question_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, question_entities: Optional[List[MessageEntity]] = None, is_anonymous: Optional[bool] = None, type: Optional[str] = None, allows_multiple_answers: Optional[bool] = None, correct_option_id: Optional[int] = None, explanation: Optional[str] = None, explanation_parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, explanation_entities: Optional[List[MessageEntity]] = None, open_period: Optional[int] = None, close_date: Optional[Union[datetime.datetime, datetime.timedelta, int]] = None, is_closed: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendPoll*

Shortcut for method `aiogram.methods.send_poll.SendPoll` will automatically fill method attributes:

- `chat_id`

- `message_thread_id`
- `business_connection_id`

Use this method to send a native poll. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpoll

Parameters

- **question** – Poll question, 1-300 characters
- **options** – A JSON-serialized list of 2-10 answer options
- **question_parse_mode** – Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed
- **question_entities** – A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*
- **is_anonymous** – `True`, if the poll needs to be anonymous, defaults to `True`
- **type** – Poll type, 'quiz' or 'regular', defaults to 'regular'
- **allows_multiple_answers** – `True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`
- **correct_option_id** – 0-based identifier of the correct answer option, required for polls in quiz mode
- **explanation** – Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing
- **explanation_parse_mode** – Mode for parsing entities in the explanation. See formatting options for more details.
- **explanation_entities** – A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*
- **open_period** – Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.
- **close_date** – Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.
- **is_closed** – Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_poll.SendPoll`

reply_dice(*emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → SendDice

Shortcut for method `aiogram.methods.send_dice.SendDice` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send an animated emoji that will display a random value. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#senddice

Parameters

- **emoji** – Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_dice.SendDice`

answer_dice(*emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → SendDice

Shortcut for method `aiogram.methods.send_dice.SendDice` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`

- `business_connection_id`

Use this method to send an animated emoji that will display a random value. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#senddice

Parameters

- **emoji** – Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_dice.SendDice`

reply_sticker(*sticker: Union[InputFile, str], emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendSticker*

Shortcut for method `aiogram.methods.send_sticker.SendSticker` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendsticker

Parameters

- **sticker** – Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.

- **emoji** – Emoji associated with the sticker; only for just uploaded stickers
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method *aiogram.methods.send_sticker.SendSticker*

answer_sticker(*sticker: Union[InputFile, str], emoji: Optional[str] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendSticker*

Shortcut for method *aiogram.methods.send_sticker.SendSticker* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendsticker

Parameters

- **sticker** – Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.
- **emoji** – Emoji associated with the sticker; only for just uploaded stickers
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_sticker.SendSticker`

reply_venue(*latitude: float*, *longitude: float*, *title: str*, *address: str*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, ***kwargs: Any*) → *SendVenue*

Shortcut for method `aiogram.methods.send_venue.SendVenue` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

Parameters

- **latitude** – Latitude of the venue
- **longitude** – Longitude of the venue
- **title** – Name of the venue
- **address** – Address of the venue
- **foursquare_id** – Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)
- **google_place_id** – Google Places identifier of the venue
- **google_place_type** – Google Places type of the venue. (See supported types.)
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_venue.SendVenue`

answer_venue(*latitude: float*, *longitude: float*, *title: str*, *address: str*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>*, *message_effect_id: Optional[str] = None*, *reply_parameters: Optional[ReplyParameters] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, ***kwargs: Any*) → SendVenue

Shortcut for method `aiogram.methods.send_venue.SendVenue` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

Parameters

- **latitude** – Latitude of the venue
- **longitude** – Longitude of the venue
- **title** – Name of the venue
- **address** – Address of the venue
- **foursquare_id** – Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)
- **google_place_id** – Google Places identifier of the venue
- **google_place_type** – Google Places type of the venue. (See supported types.)
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns
instance of method `aiogram.methods.send_venue.SendVenue`

reply_video(*video: Union[InputFile, str], duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, supports_streaming: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendVideo*

Shortcut for method `aiogram.methods.send_video.SendVideo` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

Parameters

- **video** – Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media

- **has_spoiler** – Pass `True` if the video needs to be covered with a spoiler animation
- **supports_streaming** – Pass `True` if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns
instance of method `aiogram.methods.send_video.SendVideo`

answer_video(*video: Union[InputFile, str], duration: Optional[int] = None, width: Optional[int] = None, height: Optional[int] = None, thumbnail: Optional[InputFile] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, has_spoiler: Optional[bool] = None, supports_streaming: Optional[bool] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideo*

Shortcut for method `aiogram.methods.send_video.SendVideo` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

Parameters

- **video** – Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height

- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **has_spoiler** – Pass `True` if the video needs to be covered with a spoiler animation
- **supports_streaming** – Pass `True` if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_video.SendVideo`

reply_video_note(*video_note: Union[InputFile, str], duration: Optional[int] = None, length: Optional[int] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendVideoNote*

Shortcut for method `aiogram.methods.send_video_note.SendVideoNote` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **video_note** – Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported
- **duration** – Duration of sent video in seconds
- **length** – Video width and height, i.e. diameter of the video message
- **thumbnail** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method *aiogram.methods.send_video_note.SendVideoNote*

answer_video_note(*video_note: Union[InputFile, str], duration: Optional[int] = None, length: Optional[int] = None, thumbnail: Optional[InputFile] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVideoNote*

Shortcut for method *aiogram.methods.send_video_note.SendVideoNote* will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

Parameters

- `video_note` – Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported
- `duration` – Duration of sent video in seconds
- `length` – Video width and height, i.e. diameter of the video message
- `thumbnail` – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- `disable_notification` – Sends the message silently. Users will receive a notification with no sound.
- `protect_content` – Protects the contents of the sent message from forwarding and saving
- `message_effect_id` – Unique identifier of the message effect to be added to the message; for private chats only
- `reply_parameters` – Description of the message to reply to
- `reply_markup` – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- `allow_sending_without_reply` – Pass `True` if the message should be sent even if the specified replied-to message is not found
- `reply_to_message_id` – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_video_note.SendVideoNote`

`reply_voice`(*voice: Union[InputFile, str], caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, **kwargs: Any*) → *SendVoice*

Shortcut for method `aiogram.methods.send_voice.SendVoice` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as `aiogram.types.audio.Audio` or `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

Parameters

- **voice** – Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **caption** – Voice message caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the voice message caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – Duration of the voice message in seconds
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **message_effect_id** – Unique identifier of the message effect to be added to the message; for private chats only
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found

Returns

instance of method `aiogram.methods.send_voice.SendVoice`

answer_voice(*voice: Union[InputFile, str], caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, duration: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, message_effect_id: Optional[str] = None, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *SendVoice*

Shortcut for method `aiogram.methods.send_voice.SendVoice` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as `aiogram.types.audio.Audio` or `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

Parameters

- **`voice`** – Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*
- **`caption`** – Voice message caption, 0-1024 characters after entities parsing
- **`parse_mode`** – Mode for parsing entities in the voice message caption. See formatting options for more details.
- **`caption_entities`** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **`duration`** – Duration of the voice message in seconds
- **`disable_notification`** – Sends the message silently. Users will receive a notification with no sound.
- **`protect_content`** – Protects the contents of the sent message from forwarding and saving
- **`message_effect_id`** – Unique identifier of the message effect to be added to the message; for private chats only
- **`reply_parameters`** – Description of the message to reply to
- **`reply_markup`** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **`allow_sending_without_reply`** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **`reply_to_message_id`** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.send_voice.SendVoice`

`send_copy`(*chat_id: str | int*, *disable_notification: bool | None = None*, *reply_to_message_id: int | None = None*, *reply_parameters: ReplyParameters | None = None*, *reply_markup: InlineKeyboardMarkup | ReplyKeyboardMarkup | None = None*, *allow_sending_without_reply: bool | None = None*, *message_thread_id: int | None = None*, *business_connection_id: str | None = None*, *parse_mode: str | None = None*, *message_effect_id: str | None = None*) → *ForwardMessage* | *SendAnimation* | *SendAudio* | *SendContact* | *SendDocument* | *SendLocation* | *SendMessage* | *SendPhoto* | *SendPoll* | *SendDice* | *SendSticker* | *SendVenue* | *SendVideo* | *SendVideoNote* | *SendVoice*

Send copy of a message.

Is similar to `aiogram.client.bot.Bot.copy_message()` but returning the sent message instead of `aiogram.types.message_id.MessageId`

Note

This method doesn't use the API method named *copyMessage* and historically implemented before the similar method is added to API

Parameters

- **chat_id**
- **disable_notification**
- **reply_to_message_id**
- **reply_parameters**
- **reply_markup**
- **allow_sending_without_reply**
- **message_thread_id**
- **business_connection_id**
- **parse_mode**
- **message_effect_id**

Returns

copy_to(*chat_id: Union[int, str], message_thread_id: Optional[int] = None, caption: Optional[str] = None, parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>, caption_entities: Optional[List[MessageEntity]] = None, show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, reply_parameters: Optional[ReplyParameters] = None, reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None, allow_sending_without_reply: Optional[bool] = None, reply_to_message_id: Optional[int] = None, **kwargs: Any*) → *CopyMessage*

Shortcut for method `aiogram.methods.copy_message.CopyMessage` will automatically fill method attributes:

- `from_chat_id`
- `message_id`

Use this method to copy messages of any kind. Service messages, paid media messages, giveaway messages, giveaway winners messages, and invoice messages can't be copied. A quiz `aiogram.methods.poll.Poll` can be copied only if the value of the field *correct_option_id* is known to the bot. The method is analogous to the method `aiogram.methods.forward_message.ForwardMessage`, but the copied message doesn't have a link to the original message. Returns the `aiogram.types.message_id.MessageId` of the sent message on success.

Source: https://core.telegram.org/bots/api#copymessage

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **caption** – New caption for media, 0-1024 characters after entities parsing. If not specified, the original caption is kept
- **parse_mode** – Mode for parsing entities in the new caption. See formatting options for more details.

- **caption_entities** – A JSON-serialized list of special entities that appear in the new caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media. Ignored if a new caption isn't specified.
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user
- **allow_sending_without_reply** – Pass `True` if the message should be sent even if the specified replied-to message is not found
- **reply_to_message_id** – If the message is a reply, ID of the original message

Returns

instance of method `aiogram.methods.copy_message.CopyMessage`

edit_text(*text: str*, *business_connection_id: Optional[str] = None*, *inline_message_id: Optional[str] = None*, *parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>*, *entities: Optional[List[MessageEntity]] = None*, *link_preview_options: Optional[Union[LinkPreviewOptions, Default]] = <Default('link_preview')>*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, *disable_web_page_preview: Optional[Union[bool, Default]] = <Default('link_preview_is_disabled')>*, ***kwargs: Any*) → *EditMessageText*

Shortcut for method `aiogram.methods.edit_message_text.EditMessageText` will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to edit text and game messages. On success, if the edited message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagetext

Parameters

- **text** – New text of the message, 1-4096 characters after entities parsing
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message to be edited was sent
- **inline_message_id** – Required if *chat_id* and *message_id* are not specified. Identifier of the inline message
- **parse_mode** – Mode for parsing entities in the message text. See formatting options for more details.
- **entities** – A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*
- **link_preview_options** – Link preview generation options for the message
- **reply_markup** – A JSON-serialized object for an inline keyboard.

- **disable_web_page_preview** – Disables link previews for links in this message

Returns
instance of method *aiogram.methods.edit_message_text.EditMessageText*

forward(*chat_id: Union[int, str], message_thread_id: Optional[int] = None, disable_notification: Optional[bool] = None, protect_content: Optional[Union[bool, Default]] = <Default('protect_content')>, **kwargs: Any*) → *ForwardMessage*

Shortcut for method *aiogram.methods.forward_message.ForwardMessage* will automatically fill method attributes:

- `from_chat_id`
- `message_id`

Use this method to forward messages of any kind. Service messages and messages with protected content can't be forwarded. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#forwardmessage

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)
- **message_thread_id** – Unique identifier for the target message thread (topic) of the forum; for forum supergroups only
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the forwarded message from forwarding and saving

Returns
instance of method *aiogram.methods.forward_message.ForwardMessage*

edit_media(*media:* InputMediaAnimation | InputMediaDocument | InputMediaAudio | InputMediaPhoto | InputMediaVideo, *business_connection_id: str* | *None = None*, *inline_message_id: str* | *None = None*, *reply_markup:* InlineKeyboardMarkup | *None = None*, ***kwargs: Any*) → *EditMessageMedia*

Shortcut for method *aiogram.methods.edit_message_media.EditMessageMedia* will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to edit animation, audio, document, photo, or video messages. If a message is part of a message album, then it can be edited only to an audio for audio albums, only to a document for document albums and to a photo or a video otherwise. When an inline message is edited, a new file can't be uploaded; use a previously uploaded file via its file_id or specify a URL. On success, if the edited message is not an inline message, the edited *aiogram.types.message.Message* is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagemedia

Parameters

- **media** – A JSON-serialized object for a new media content of the message

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message to be edited was sent
- **inline_message_id** – Required if *chat_id* and *message_id* are not specified. Identifier of the inline message
- **reply_markup** – A JSON-serialized object for a new inline keyboard.

Returns
instance of method `aiogram.methods.edit_message_media.EditMessageMedia`

edit_reply_markup(*business_connection_id: str | None = None*, *inline_message_id: str | None = None*, *reply_markup:* InlineKeyboardMarkup *| None = None*, ***kwargs: Any*) → *EditMessageReplyMarkup*

Shortcut for method `aiogram.methods.edit_message_reply_markup.EditMessageReplyMarkup` will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to edit only the reply markup of messages. On success, if the edited message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagereplymarkup

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message to be edited was sent
- **inline_message_id** – Required if *chat_id* and *message_id* are not specified. Identifier of the inline message
- **reply_markup** – A JSON-serialized object for an inline keyboard.

Returns
instance of method `aiogram.methods.edit_message_reply_markup.EditMessageReplyMarkup`

delete_reply_markup(*business_connection_id: str | None = None*, *inline_message_id: str | None = None*, ***kwargs: Any*) → *EditMessageReplyMarkup*

Shortcut for method `aiogram.methods.edit_message_reply_markup.EditMessageReplyMarkup` will automatically fill method attributes:

- `chat_id`
- `message_id`
- `reply_markup`

Use this method to edit only the reply markup of messages. On success, if the edited message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagereplymarkup

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message to be edited was sent

- **inline_message_id** – Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

Returns
instance of method `aiogram.methods.edit_message_reply_markup.EditMessageReplyMarkup`

edit_live_location(*latitude: float*, *longitude: float*, *business_connection_id: str | None = None*, *inline_message_id: str | None = None*, *live_period: int | None = None*, *horizontal_accuracy: float | None = None*, *heading: int | None = None*, *proximity_alert_radius: int | None = None*, *reply_markup:* InlineKeyboardMarkup *| None = None*, ***kwargs: Any*) → *EditMessageLiveLocation*

Shortcut for method `aiogram.methods.edit_message_live_location.EditMessageLiveLocation` will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to edit live location messages. A location can be edited until its *live_period* expires or editing is explicitly disabled by a call to `aiogram.methods.stop_message_live_location.StopMessageLiveLocation`. On success, if the edited message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned.

Source: https://core.telegram.org/bots/api#editmessagelivelocation

Parameters

- **latitude** – Latitude of new location
- **longitude** – Longitude of new location
- **business_connection_id** – Unique identifier of the business connection on behalf of which the message to be edited was sent
- **inline_message_id** – Required if *chat_id* and *message_id* are not specified. Identifier of the inline message
- **live_period** – New period in seconds during which the location can be updated, starting from the message send date. If 0x7FFFFFFF is specified, then the location can be updated forever. Otherwise, the new value must not exceed the current *live_period* by more than a day, and the live location expiration date must remain within the next 90 days. If not specified, then *live_period* remains unchanged
- **horizontal_accuracy** – The radius of uncertainty for the location, measured in meters; 0-1500
- **heading** – Direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.
- **proximity_alert_radius** – The maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.
- **reply_markup** – A JSON-serialized object for a new inline keyboard.

Returns
instance of method `aiogram.methods.edit_message_live_location.EditMessageLiveLocation`

stop_live_location(*business_connection_id: str | None = None*, *inline_message_id: str | None = None*, *reply_markup:* InlineKeyboardMarkup *| None = None*, ***kwargs: Any*) → *StopMessageLiveLocation*

Shortcut for method `aiogram.methods.stop_message_live_location.StopMessageLiveLocation` will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to stop updating a live location message before *live_period* expires. On success, if the message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned.

Source: https://core.telegram.org/bots/api#stopmessagelivelocation

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message to be edited was sent
- **inline_message_id** – Required if *chat_id* and *message_id* are not specified. Identifier of the inline message
- **reply_markup** – A JSON-serialized object for a new inline keyboard.

Returns

instance of method `aiogram.methods.stop_message_live_location.StopMessageLiveLocation`

edit_caption(*business_connection_id: Optional[str] = None*, *inline_message_id: Optional[str] = None*, *caption: Optional[str] = None*, *parse_mode: Optional[Union[str, Default]] = <Default('parse_mode')>*, *caption_entities: Optional[List[MessageEntity]] = None*, *show_caption_above_media: Optional[Union[bool, Default]] = <Default('show_caption_above_media')>*, *reply_markup: Optional[InlineKeyboardMarkup] = None*, ***kwargs: Any*) → *EditMessageCaption*

Shortcut for method `aiogram.methods.edit_message_caption.EditMessageCaption` will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to edit captions of messages. On success, if the edited message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagecaption

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message to be edited was sent
- **inline_message_id** – Required if *chat_id* and *message_id* are not specified. Identifier of the inline message
- **caption** – New caption of the message, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the message caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media. Supported only for animation, photo and video messages.
- **reply_markup** – A JSON-serialized object for an inline keyboard.

Returns
instance of method *aiogram.methods.edit_message_caption.EditMessageCaption*

delete(***kwargs: Any*) → *DeleteMessage*

Shortcut for method *aiogram.methods.delete_message.DeleteMessage* will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to delete a message, including service messages, with the following limitations:

- A message can only be deleted if it was sent less than 48 hours ago.
- Service messages about a supergroup, channel, or forum topic creation can't be deleted.
- A dice message in a private chat can only be deleted if it was sent more than 24 hours ago.
- Bots can delete outgoing messages in private chats, groups, and supergroups.
- Bots can delete incoming messages in private chats.
- Bots granted *can_post_messages* permissions can delete outgoing messages in channels.
- If the bot is an administrator of a group, it can delete any message there.
- If the bot has *can_delete_messages* permission in a supergroup or a channel, it can delete any message there.

Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletemessage

Returns
instance of method *aiogram.methods.delete_message.DeleteMessage*

pin(*business_connection_id: str | None = None*, *disable_notification: bool | None = None*, ***kwargs: Any*) → *PinChatMessage*

Shortcut for method *aiogram.methods.pin_chat_message.PinChatMessage* will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to add a message to the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#pinchatmessage

Parameters

- **business_connection_id** – Unique identifier of the business connection on behalf of which the message will be pinned
- **disable_notification** – Pass `True` if it is not necessary to send a notification to all chat members about the new pinned message. Notifications are always disabled in channels and private chats.

Returns
instance of method *aiogram.methods.pin_chat_message.PinChatMessage*

unpin(*business_connection_id: str | None = None*, ***kwargs: Any*) → *UnpinChatMessage*

Shortcut for method *aiogram.methods.unpin_chat_message.UnpinChatMessage* will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to remove a message from the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unpinchatmessage

Parameters
business_connection_id – Unique identifier of the business connection on behalf of which the message will be unpinned

Returns
instance of method *aiogram.methods.unpin_chat_message.UnpinChatMessage*

get_url(*force_private: bool = False*, *include_thread_id: bool = False*) → str | None

Returns message URL. Cannot be used in private (one-to-one) chats. If chat has a username, returns URL like https://t.me/username/message_id Otherwise (or if {force_private} flag is set), returns https://t.me/c/shifted_chat_id/message_id

Parameters

- **force_private** – if set, a private URL is returned even for a public chat
- **include_thread_id** – if set, adds chat thread id to URL and returns like https://t.me/username/thread_id/message_id

Returns
string with full message URL

react(*reaction: List[ReactionTypeEmoji | ReactionTypeCustomEmoji | ReactionTypePaid] | None = None*, *is_big: bool | None = None*, ***kwargs: Any*) → *SetMessageReaction*

Shortcut for method *aiogram.methods.set_message_reaction.SetMessageReaction* will automatically fill method attributes:

- `chat_id`
- `message_id`

Use this method to change the chosen reactions on a message. Service messages can't be reacted to. Automatically forwarded messages from a channel to its discussion group have the same available reactions as messages in the channel. Bots can't use paid reactions. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setmessagereaction

Parameters

- **reaction** – A JSON-serialized list of reaction types to set on the message. Currently, as non-premium users, bots can set up to one reaction per message. A custom emoji reaction can be used if it is either already present on the message or explicitly allowed by chat administrators. Paid reactions can't be used by bots.
- **is_big** – Pass `True` to set the reaction with a big animation

Returns
instance of method `aiogram.methods.set_message_reaction.SetMessageReaction`

answer_paid_media(*star_count: int*, *media: List[InputPaidMediaPhoto | InputPaidMediaVideo]*, *payload: str | None = None*, *caption: str | None = None*, *parse_mode: str | None = None*, *caption_entities: List[MessageEntity] | None = None*, *show_caption_above_media: bool | None = None*, *disable_notification: bool | None = None*, *protect_content: bool | None = None*, *reply_parameters: ReplyParameters | None = None*, *reply_markup: InlineKeyboardMarkup | ReplyKeyboardMarkup | ReplyKeyboardRemove | ForceReply | None = None*, ***kwargs: Any*) → *SendPaidMedia*

Shortcut for method `aiogram.methods.send_paid_media.SendPaidMedia` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`

Use this method to send paid media. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpaidmedia

Parameters

- **star_count** – The number of Telegram Stars that must be paid to buy access to the media; 1-2500
- **media** – A JSON-serialized array describing the media to be sent; up to 10 items
- **payload** – Bot-defined paid media payload, 0-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **caption** – Media caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the media caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **reply_parameters** – Description of the message to reply to
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

Returns
instance of method `aiogram.methods.send_paid_media.SendPaidMedia`

reply_paid_media(*star_count: int*, *media: List[*InputPaidMediaPhoto | InputPaidMediaVideo*]*, *payload: str | None = None*, *caption: str | None = None*, *parse_mode: str | None = None*, *caption_entities: List[*MessageEntity*] | None = None*, *show_caption_above_media: bool | None = None*, *disable_notification: bool | None = None*, *protect_content: bool | None = None*, *reply_markup:* InlineKeyboardMarkup | ReplyKeyboardMarkup | ReplyKeyboardRemove | ForceReply | *None = None*, ***kwargs: Any*) → *SendPaidMedia*

Shortcut for method `aiogram.methods.send_paid_media.SendPaidMedia` will automatically fill method attributes:

- `chat_id`
- `message_thread_id`
- `business_connection_id`
- `reply_parameters`

Use this method to send paid media. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpaidmedia

Parameters

- **star_count** – The number of Telegram Stars that must be paid to buy access to the media; 1-2500
- **media** – A JSON-serialized array describing the media to be sent; up to 10 items
- **payload** – Bot-defined paid media payload, 0-128 bytes. This will not be displayed to the user, use it for your internal processes.
- **caption** – Media caption, 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the media caption. See formatting options for more details.
- **caption_entities** – A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **show_caption_above_media** – Pass `True`, if the caption must be shown above the message media
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – Protects the contents of the sent message from forwarding and saving
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

Returns

instance of method `aiogram.methods.send_paid_media.SendPaidMedia`

MessageAutoDeleteTimerChanged

class aiogram.types.message_auto_delete_timer_changed.**MessageAutoDeleteTimerChanged**(*, *message_auto_delete_time: int*, ***extra_data: Any*)

This object represents a service message about a change in auto-delete timer settings.

Source: https://core.telegram.org/bots/api#messageautodeletetimerchanged

message_auto_delete_time: int

New auto-delete time for messages in the chat; in seconds

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

MessageEntity

class aiogram.types.message_entity.**MessageEntity**(*, *type: str*, *offset: int*, *length: int*, *url: str | None = None*, *user: User | None = None*, *language: str | None = None*, *custom_emoji_id: str | None = None*, ***extra_data: Any*)

This object represents one special entity in a text message. For example, hashtags, usernames, URLs, etc.

Source: https://core.telegram.org/bots/api#messageentity

type: str

Type of the entity. Currently, can be 'mention' (`@username`), 'hashtag' (`#hashtag`), 'cashtag' (`$USD`), 'bot_command' (`/start@jobs_bot`), 'url' (`https://telegram.org`), 'email' (`do-not-reply@telegram.org`), 'phone_number' (`+1-212-555-0123`), 'bold' (**bold text**), 'italic' (*italic text*), 'underline' (underlined text), 'strikethrough' (strikethrough text), 'spoiler' (spoiler message), 'blockquote' (block quotation), 'expandable_blockquote' (collapsed-by-default block quotation), 'code' (monowidth string), 'pre' (monowidth block), 'text_link' (for clickable text URLs), 'text_mention' (for users without usernames), 'custom_emoji' (for inline custom emoji stickers)

offset: int

Offset in UTF-16 code units to the start of the entity

length: int

Length of the entity in UTF-16 code units

url: str | None

Optional. For 'text_link' only, URL that will be opened after user taps on the text

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user: *User* | None

Optional. For 'text_mention' only, the mentioned user

language: str | None

Optional. For 'pre' only, the programming language of the entity text

custom_emoji_id: str | None

Optional. For 'custom_emoji' only, unique identifier of the custom emoji. Use *aiogram.methods.get_custom_emoji_stickers.GetCustomEmojiStickers* to get full information about the sticker

extract_from(*text: str*) → str

MessageId

class aiogram.types.message_id.**MessageId**(*, *message_id: int*, ***extra_data: Any*)

This object represents a unique message identifier.

Source: https://core.telegram.org/bots/api#messageid

message_id: int

Unique message identifier

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

MessageOrigin

class aiogram.types.message_origin.**MessageOrigin**(***extra_data: Any*)

This object describes the origin of a message. It can be one of

- *aiogram.types.message_origin_user.MessageOriginUser*
- *aiogram.types.message_origin_hidden_user.MessageOriginHiddenUser*
- *aiogram.types.message_origin_chat.MessageOriginChat*
- *aiogram.types.message_origin_channel.MessageOriginChannel*

Source: https://core.telegram.org/bots/api#messageorigin

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

MessageOriginChannel

class aiogram.types.message_origin_channel.**MessageOriginChannel**(*, *type: Literal[MessageOriginType.CHANNEL] = MessageOriginType.CHANNEL*, *date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *chat: Chat*, *message_id: int*, *author_signature: str | None = None*, ***extra_data: Any*)

The message was originally sent to a channel chat.

Source: https://core.telegram.org/bots/api#messageoriginchannel

type: Literal[MessageOriginType.CHANNEL]

Type of the message origin, always 'channel'

date: DateTime

Date the message was sent originally in Unix time

chat: *Chat*

Channel chat to which the message was originally sent

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_id: int

Unique message identifier inside the chat

author_signature: str | None

Optional. Signature of the original post author

MessageOriginChat

class aiogram.types.message_origin_chat.**MessageOriginChat**(*, *type: Literal[MessageOriginType.CHAT] = MessageOriginType.CHAT*, *date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *sender_chat: Chat*, *author_signature: str | None = None*, ***extra_data: Any*)

The message was originally sent on behalf of a chat to a group chat.

Source: https://core.telegram.org/bots/api#messageoriginchat

type: Literal[MessageOriginType.CHAT]

Type of the message origin, always 'chat'

date: DateTime

Date the message was sent originally in Unix time

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sender_chat: *Chat*

Chat that sent the message originally

author_signature: str | None

Optional. For messages originally sent by an anonymous chat administrator, original message author signature

MessageOriginHiddenUser

class aiogram.types.message_origin_hidden_user.**MessageOriginHiddenUser**(*, *type: Literal[MessageOriginType.HIDDEN_USER = MessageOriginType.HIDDEN_USER*, *date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *sender_user_name: str*, ***extra_data: Any*)

The message was originally sent by an unknown user.

Source: https://core.telegram.org/bots/api#messageoriginhiddenuser

type: Literal[MessageOriginType.HIDDEN_USER]

Type of the message origin, always 'hidden_user'

date: _datetime_serializer, return_type=int, when_used=unless-none)]

Date the message was sent originally in Unix time

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sender_user_name: str

Name of the user that sent the message originally

MessageOriginUser

class aiogram.types.message_origin_user.**MessageOriginUser**(*, *type: Literal[MessageOriginType.USER] = MessageOriginType.USER*, *date: _datetime_serializer, return_type=int, when_used=unless - none)]*, *sender_user:* User, ***extra_data: Any*)

The message was originally sent by a known user.

Source: https://core.telegram.org/bots/api#messageoriginuser

type: Literal[MessageOriginType.USER]

Type of the message origin, always 'user'

date: DateTime

Date the message was sent originally in Unix time

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sender_user: ***User***

User that sent the message originally

MessageReactionCountUpdated

class aiogram.types.message_reaction_count_updated.**MessageReactionCountUpdated**(**, chat: Chat, message_id: int, date: _datetime_serializer, return_type=int, when_used=unless - none)], reactions: List[ReactionCount], **extra_data: Any*)

This object represents reaction changes on a message with anonymous reactions.

Source: https://core.telegram.org/bots/api#messagereactioncountupdated

chat: ***Chat***

The chat containing the message

message_id: int

Unique message identifier inside the chat

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

date: DateTime

Date of the change in Unix time

reactions: List[*ReactionCount*]

List of reactions that are present on the message

MessageReactionUpdated

class aiogram.types.message_reaction_updated.**MessageReactionUpdated**(**, chat: Chat, message_id: int, date: _datetime_serializer, return_type=int, when_used=unless - none)], old_reaction: List[ReactionTypeEmoji | ReactionTypeCustomEmoji | ReactionTypePaid], new_reaction: List[ReactionTypeEmoji | ReactionTypeCustomEmoji | ReactionTypePaid], user: User | None = None, actor_chat: Chat | None = None, **extra_data: Any*)

This object represents a change of a reaction on a message performed by a user.

Source: https://core.telegram.org/bots/api#messagereactionupdated

chat: *Chat*

The chat containing the message the user reacted to

message_id: **int**

Unique identifier of the message inside the chat

date: **DateTime**

Date of the change in Unix time

old_reaction: **List[***ReactionTypeEmoji* **|** *ReactionTypeCustomEmoji* **|** *ReactionTypePaid***]**

Previous list of reaction types that were set by the user

model_computed_fields: **ClassVar[Dict[str, ComputedFieldInfo]] = {}**

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

new_reaction: **List[***ReactionTypeEmoji* **|** *ReactionTypeCustomEmoji* **|** *ReactionTypePaid***]**

New list of reaction types that have been set by the user

user: *User* **| None**

Optional. The user that changed the reaction, if the user isn't anonymous

actor_chat: *Chat* **| None**

Optional. The chat on behalf of which the reaction was changed, if the user is anonymous

PaidMedia

class aiogram.types.paid_media.**PaidMedia**(***extra_data: Any*)

This object describes paid media. Currently, it can be one of

- *aiogram.types.paid_media_preview.PaidMediaPreview*
- *aiogram.types.paid_media_photo.PaidMediaPhoto*
- *aiogram.types.paid_media_video.PaidMediaVideo*

Source: https://core.telegram.org/bots/api#paidmedia

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

PaidMediaInfo

class aiogram.types.paid_media_info.**PaidMediaInfo**(**, star_count: int, paid_media: List[PaidMediaPreview | PaidMediaPhoto | PaidMediaVideo], **extra_data: Any*)

Describes the paid media added to a message.

Source: https://core.telegram.org/bots/api#paidmediainfo

star_count: int

The number of Telegram Stars that must be paid to buy access to the media

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

paid_media: List[*PaidMediaPreview* | *PaidMediaPhoto* | *PaidMediaVideo*]

Information about the paid media

PaidMediaPhoto

class aiogram.types.paid_media_photo.**PaidMediaPhoto**(**, type: Literal[PaidMediaType.PHOTO] = PaidMediaType.PHOTO, photo: List[PhotoSize], **extra_data: Any*)

The paid media is a photo.

Source: https://core.telegram.org/bots/api#paidmediaphoto

type: Literal[PaidMediaType.PHOTO]

Type of the paid media, always 'photo'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

photo: List[*PhotoSize*]

The photo

PaidMediaPreview

class aiogram.types.paid_media_preview.**PaidMediaPreview**(**, type: Literal[PaidMediaType.PREVIEW] = PaidMediaType.PREVIEW, width: int | None = None, height: int | None = None, duration: int | None = None, **extra_data: Any*)

The paid media isn't available before the payment.

Source: https://core.telegram.org/bots/api#paidmediapreview

type: Literal[PaidMediaType.PREVIEW]

Type of the paid media, always 'preview'

width: int | None

Optional. Media width as defined by the sender

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

height: int | None

Optional. Media height as defined by the sender

duration: int | None

Optional. Duration of the media in seconds as defined by the sender

PaidMediaVideo

class aiogram.types.paid_media_video.**PaidMediaVideo**(**, type: Literal[PaidMediaType.VIDEO] = PaidMediaType.VIDEO, video: Video, **extra_data: Any*)

The paid media is a video.

Source: https://core.telegram.org/bots/api#paidmediavideo

type: Literal[PaidMediaType.VIDEO]

Type of the paid media, always 'video'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

video: *Video*

The video

PhotoSize

class aiogram.types.photo_size.**PhotoSize**(**, file_id: str, file_unique_id: str, width: int, height: int, file_size: int | None = None, **extra_data: Any*)

This object represents one size of a photo or a file / `aiogram.methods.sticker.Sticker` thumbnail.

Source: https://core.telegram.org/bots/api#photosize

file_id: str

Identifier for this file, which can be used to download or reuse the file

file_unique_id: str

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

width: int

Photo width

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

height: int

Photo height

file_size: int | None

Optional. File size in bytes

Poll

class aiogram.types.poll.**Poll**(**, id: str, question: str, options: List[PollOption], total_voter_count: int, is_closed: bool, is_anonymous: bool, type: str, allows_multiple_answers: bool, question_entities: List[MessageEntity] | None = None, correct_option_id: int | None = None, explanation: str | None = None, explanation_entities: List[MessageEntity] | None = None, open_period: int | None = None, close_date: _datetime_serializer, return_type=int, when_used=unless - none)] | None = None, **extra_data: Any*)

This object contains information about a poll.

Source: https://core.telegram.org/bots/api#poll

id: str

Unique poll identifier

question: str

Poll question, 1-300 characters

options: List[*PollOption*]

List of poll options

total_voter_count: int

Total number of users that voted in the poll

is_closed: bool

True, if the poll is closed

is_anonymous: bool

True, if the poll is anonymous

type: str

Poll type, currently can be 'regular' or 'quiz'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

allows_multiple_answers: bool

True, if the poll allows multiple answers

question_entities: List[*MessageEntity*] | None

Optional. Special entities that appear in the *question*. Currently, only custom emoji entities are allowed in poll questions

correct_option_id: int | None

Optional. 0-based identifier of the correct answer option. Available only for polls in the quiz mode, which are closed, or was sent (not forwarded) by the bot or to the private chat with the bot.

explanation: str | None

Optional. Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters

explanation_entities: List[*MessageEntity*] | None

Optional. Special entities like usernames, URLs, bot commands, etc. that appear in the *explanation*

open_period: int | None

Optional. Amount of time in seconds the poll will be active after creation

close_date: DateTime | None

Optional. Point in time (Unix timestamp) when the poll will be automatically closed

PollAnswer

class aiogram.types.poll_answer.**PollAnswer**(**, poll_id: str, option_ids: List[int], voter_chat: Chat | None = None, user: User | None = None, **extra_data: Any*)

This object represents an answer of a user in a non-anonymous poll.

Source: https://core.telegram.org/bots/api#pollanswer

poll_id: str

Unique poll identifier

option_ids: List[int]

0-based identifiers of chosen answer options. May be empty if the vote was retracted.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

voter_chat: *Chat* **| None**

Optional. The chat that changed the answer to the poll, if the voter is anonymous

user: *User* **| None**

Optional. The user that changed the answer to the poll, if the voter isn't anonymous

PollOption

class aiogram.types.poll_option.**PollOption**(***, *text: str*, *voter_count: int*, *text_entities: List[MessageEntity] | None = None*, ***extra_data: Any*)

This object contains information about one answer option in a poll.

Source: https://core.telegram.org/bots/api#polloption

text: str

Option text, 1-100 characters

voter_count: int

Number of users that voted for this option

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

text_entities: List[*MessageEntity*] | None

Optional. Special entities that appear in the option *text*. Currently, only custom emoji entities are allowed in poll option texts

ProximityAlertTriggered

class aiogram.types.proximity_alert_triggered.**ProximityAlertTriggered**(***, *traveler: User*, *watcher: User*, *distance: int*, ***extra_data: Any*)

This object represents the content of a service message, sent whenever a user in the chat triggers a proximity alert set by another user.

Source: https://core.telegram.org/bots/api#proximityalerttriggered

traveler: *User*

User that triggered the alert

watcher: *User*

User that set the alert

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

distance: int

The distance between the users

ReactionCount

class aiogram.types.reaction_count.**ReactionCount**(*, *type: ReactionTypeEmoji | ReactionTypeCustomEmoji | ReactionTypePaid*, *total_count: int*, ***extra_data: Any*)

Represents a reaction added to a message along with the number of times it was added.

Source: https://core.telegram.org/bots/api#reactioncount

type: *ReactionTypeEmoji* **|** *ReactionTypeCustomEmoji* **|** *ReactionTypePaid*

Type of the reaction

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

total_count: int

Number of times the reaction was added

ReactionType

class aiogram.types.reaction_type.**ReactionType**(***extra_data: Any*)

This object describes the type of a reaction. Currently, it can be one of

- *aiogram.types.reaction_type_emoji.ReactionTypeEmoji*
- *aiogram.types.reaction_type_custom_emoji.ReactionTypeCustomEmoji*
- *aiogram.types.reaction_type_paid.ReactionTypePaid*

Source: https://core.telegram.org/bots/api#reactiontype

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ReactionTypeCustomEmoji

class aiogram.types.reaction_type_custom_emoji.**ReactionTypeCustomEmoji**(*, *type: Literal[ReactionTypeType.CUSTOM_EMOJI = ReactionTypeType.CUSTOM_EMOJI*, *custom_emoji_id: str*, ***extra_data: Any*)

The reaction is based on a custom emoji.

Source: https://core.telegram.org/bots/api#reactiontypecustomemoji

type: Literal[ReactionTypeType.CUSTOM_EMOJI]

Type of the reaction, always 'custom_emoji'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

custom_emoji_id: str

Custom emoji identifier

ReactionTypeEmoji

class aiogram.types.reaction_type_emoji.**ReactionTypeEmoji**(*, *type: Literal[ReactionTypeType.EMOJI] = ReactionTypeType.EMOJI*, *emoji: str*, ***extra_data: Any*)

The reaction is based on an emoji.

Source: https://core.telegram.org/bots/api#reactiontypeemoji

type: Literal[ReactionTypeType.EMOJI]

Type of the reaction, always 'emoji'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

emoji: str

Reaction emoji. Currently, it can be one of “”, “”

ReactionTypePaid

class aiogram.types.reaction_type_paid.**ReactionTypePaid**(*, *type: Literal['paid'] = 'paid'*, ***extra_data: Any*)

The reaction is paid.

Source: https://core.telegram.org/bots/api#reactiontypepaid

type: Literal['paid']

Type of the reaction, always 'paid'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ReplyKeyboardMarkup

class aiogram.types.reply_keyboard_markup.**ReplyKeyboardMarkup**(**, keyboard: List[List[KeyboardButton]], is_persistent: bool | None = None, resize_keyboard: bool | None = None, one_time_keyboard: bool | None = None, input_field_placeholder: str | None = None, selective: bool | None = None, **extra_data: Any*)

This object represents a custom keyboard with reply options (see Introduction to bots for details and examples). Not supported in channels and for messages sent on behalf of a Telegram Business account.

Source: https://core.telegram.org/bots/api#replykeyboardmarkup

keyboard: List[List[*KeyboardButton*]]

Array of button rows, each represented by an Array of `aiogram.types.keyboard_button.KeyboardButton` objects

is_persistent: bool | None

Optional. Requests clients to always show the keyboard when the regular keyboard is hidden. Defaults to *false*, in which case the custom keyboard can be hidden and opened with a keyboard icon.

resize_keyboard: bool | None

Optional. Requests clients to resize the keyboard vertically for optimal fit (e.g., make the keyboard smaller if there are just two rows of buttons). Defaults to *false*, in which case the custom keyboard is always of the same height as the app's standard keyboard.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

one_time_keyboard: bool | None

Optional. Requests clients to hide the keyboard as soon as it's been used. The keyboard will still be available, but clients will automatically display the usual letter-keyboard in the chat - the user can press a special button in the input field to see the custom keyboard again. Defaults to *false*.

input_field_placeholder: str | None

Optional. The placeholder to be shown in the input field when the keyboard is active; 1-64 characters

selective: bool | None

Optional. Use this parameter if you want to show the keyboard to specific users only. Targets: 1) users that are @mentioned in the *text* of the `aiogram.types.message.Message` object; 2) if the bot's message is a reply to a message in the same chat and forum topic, sender of the original message.

ReplyKeyboardRemove

class aiogram.types.reply_keyboard_remove.**ReplyKeyboardRemove**(*, *remove_keyboard: Literal[True] = True*, *selective: bool | None = None*, ***extra_data: Any*)

Upon receiving a message with this object, Telegram clients will remove the current custom keyboard and display the default letter-keyboard. By default, custom keyboards are displayed until a new keyboard is sent by a bot. An exception is made for one-time keyboards that are hidden immediately after the user presses a button (see `aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup`). Not supported in channels and for messages sent on behalf of a Telegram Business account.

Source: https://core.telegram.org/bots/api#replykeyboardremove

remove_keyboard: Literal[True]

Requests clients to remove the custom keyboard (user will not be able to summon this keyboard; if you want to hide the keyboard from sight but keep it accessible, use *one_time_keyboard* in `aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

selective: bool | None

Optional. Use this parameter if you want to remove the keyboard for specific users only. Targets: 1) users that are @mentioned in the *text* of the `aiogram.types.message.Message` object; 2) if the bot's message is a reply to a message in the same chat and forum topic, sender of the original message.

ReplyParameters

class aiogram.types.reply_parameters.**ReplyParameters**(*, *message_id: int*, *chat_id: int | str | None = None*, *allow_sending_without_reply: bool | ~aiogram.client.default.Default | None = <Default('allow_sending_without_reply')>*, *quote: str | None = None*, *quote_parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>*, *quote_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *quote_position: int | None = None*, ***extra_data: ~typing.Any*)

Describes reply parameters for the message that is being sent.

Source: https://core.telegram.org/bots/api#replyparameters

message_id: int

Identifier of the message that will be replied to in the current chat, or in the chat *chat_id* if it is specified

chat_id: int | str | None

Optional. If the message to be replied to is from a different chat, unique identifier for the chat or username of the channel (in the format `@channelusername`). Not supported for messages sent on behalf of a business account.

allow_sending_without_reply: bool | Default | None

Optional. Pass `True` if the message should be sent even if the specified message to be replied to is not found. Always `False` for replies in another chat or forum topic. Always `True` for messages sent on behalf of a business account.

quote: str | None

Optional. Quoted part of the message to be replied to; 0-1024 characters after entities parsing. The quote must be an exact substring of the message to be replied to, including *bold*, *italic*, *underline*, *strikethrough*, *spoiler*, and *custom_emoji* entities. The message will fail to send if the quote isn't found in the original message.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

quote_parse_mode: str | Default | None

Optional. Mode for parsing entities in the quote. See formatting options for more details.

quote_entities: List[*MessageEntity*] | None

Optional. A JSON-serialized list of special entities that appear in the quote. It can be specified instead of *quote_parse_mode*.

quote_position: int | None

Optional. Position of the quote in the original message in UTF-16 code units

ResponseParameters

class aiogram.types.response_parameters.**ResponseParameters**(**, migrate_to_chat_id: int | None = None, retry_after: int | None = None, **extra_data: Any*)

Describes why a request was unsuccessful.

Source: https://core.telegram.org/bots/api#responseparameters

migrate_to_chat_id: int | None

Optional. The group has been migrated to a supergroup with the specified identifier. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this identifier.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

retry_after: int | None

Optional. In case of exceeding flood control, the number of seconds left to wait before the request can be repeated

SharedUser

class aiogram.types.shared_user.**SharedUser**(*, *user_id: int*, *first_name: str | None = None*, *last_name: str | None = None*, *username: str | None = None*, *photo: List[PhotoSize] | None = None*, ***extra_data: Any*)

This object contains information about a user that was shared with the bot using a *aiogram.types.keyboard_button_request_users.KeyboardButtonRequestUsers* button.

Source: https://core.telegram.org/bots/api#shareduser

user_id: int

Identifier of the shared user. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so 64-bit integers or double-precision float types are safe for storing these identifiers. The bot may not have access to the user and could be unable to use this identifier, unless the user is already known to the bot by some other means.

first_name: str | None

Optional. First name of the user, if the name was requested by the bot

last_name: str | None

Optional. Last name of the user, if the name was requested by the bot

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

username: str | None

Optional. Username of the user, if the username was requested by the bot

photo: List[*PhotoSize*] | None

Optional. Available sizes of the chat photo, if the photo was requested by the bot

Story

class aiogram.types.story.**Story**(*, *chat: Chat*, *id: int*, ***extra_data: Any*)

This object represents a story.

Source: https://core.telegram.org/bots/api#story

chat: *Chat*

Chat that posted the story

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

id: int

Unique identifier for the story in the chat

SwitchInlineQueryChosenChat

class aiogram.types.switch_inline_query_chosen_chat.**SwitchInlineQueryChosenChat**(**, query: str | None = None, allow_user_chats: bool | None = None, allow_bot_chats: bool | None = None, allow_group_chats: bool | None = None, allow_channel_chats: bool | None = None, **extra_data: Any*)

This object represents an inline button that switches the current user to inline mode in a chosen chat, with an optional default inline query.

Source: https://core.telegram.org/bots/api#switchinlinequerychosenchat

query: str | None

Optional. The default inline query to be inserted in the input field. If left empty, only the bot's username will be inserted

allow_user_chats: bool | None

Optional. True, if private chats with users can be chosen

allow_bot_chats: bool | None

Optional. True, if private chats with bots can be chosen

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

allow_group_chats: bool | None

Optional. True, if group and supergroup chats can be chosen

allow_channel_chats: bool | None

Optional. True, if channel chats can be chosen

TextQuote

class `aiogram.types.text_quote.`**`TextQuote`**(***, *text: str*, *position: int*, *entities: List[*MessageEntity*] | None = None*, *is_manual: bool | None = None*, ***extra_data: Any*)

This object contains information about the quoted part of a message that is replied to by the given message.

Source: https://core.telegram.org/bots/api#textquote

`text: str`

Text of the quoted part of a message that is replied to by the given message

`position: int`

Approximate quote position in the original message in UTF-16 code units as specified by the sender

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`entities: List[`*`MessageEntity`*`] | None`

Optional. Special entities that appear in the quote. Currently, only *bold*, *italic*, *underline*, *strikethrough*, *spoiler*, and *custom_emoji* entities are kept in quotes.

`is_manual: bool | None`

Optional. True, if the quote was chosen manually by the message sender. Otherwise, the quote was added automatically by the server.

User

class `aiogram.types.user.`**`User`**(***, *id: int*, *is_bot: bool*, *first_name: str*, *last_name: str | None = None*, *username: str | None = None*, *language_code: str | None = None*, *is_premium: bool | None = None*, *added_to_attachment_menu: bool | None = None*, *can_join_groups: bool | None = None*, *can_read_all_group_messages: bool | None = None*, *supports_inline_queries: bool | None = None*, *can_connect_to_business: bool | None = None*, *has_main_web_app: bool | None = None*, ***extra_data: Any*)

This object represents a Telegram user or bot.

Source: https://core.telegram.org/bots/api#user

`id: int`

Unique identifier for this user or bot. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a 64-bit integer or double-precision float type are safe for storing this identifier.

`is_bot: bool`

`True`, if this user is a bot

`first_name: str`

User's or bot's first name

`last_name: str | None`

Optional. User's or bot's last name

username: str | None

Optional. User's or bot's username

language_code: str | None

Optional. IETF language tag of the user's language

is_premium: bool | None

Optional. `True`, if this user is a Telegram Premium user

added_to_attachment_menu: bool | None

Optional. `True`, if this user added the bot to the attachment menu

can_join_groups: bool | None

Optional. `True`, if the bot can be invited to groups. Returned only in `aiogram.methods.get_me.GetMe`.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

can_read_all_group_messages: bool | None

Optional. `True`, if privacy mode is disabled for the bot. Returned only in `aiogram.methods.get_me.GetMe`.

supports_inline_queries: bool | None

Optional. `True`, if the bot supports inline queries. Returned only in `aiogram.methods.get_me.GetMe`.

can_connect_to_business: bool | None

Optional. `True`, if the bot can be connected to a Telegram Business account to receive its messages. Returned only in `aiogram.methods.get_me.GetMe`.

has_main_web_app: bool | None

Optional. `True`, if the bot has a main Web App. Returned only in `aiogram.methods.get_me.GetMe`.

property full_name: str

property url: str

mention_markdown(*name: str | None = None*) → str

mention_html(*name: str | None = None*) → str

get_profile_photos(*offset: int | None = None*, *limit: int | None = None*, ***kwargs: Any*) → *GetUserProfilePhotos*

Shortcut for method `aiogram.methods.get_user_profile_photos.GetUserProfilePhotos` will automatically fill method attributes:

- `user_id`

Use this method to get a list of profile pictures for a user. Returns a `aiogram.types.user_profile_photos.UserProfilePhotos` object.

Source: https://core.telegram.org/bots/api#getuserprofilephotos

Parameters

- **offset** – Sequential number of the first photo to be returned. By default, all photos are returned.

- **limit** – Limits the number of photos to be retrieved. Values between 1-100 are accepted. Defaults to 100.

Returns
instance of method *aiogram.methods.get_user_profile_photos.GetUserProfilePhotos*

UserChatBoosts

class aiogram.types.user_chat_boosts.**UserChatBoosts**(**, boosts: List[ChatBoost], **extra_data: Any*)

This object represents a list of boosts added to a chat by a user.

Source: https://core.telegram.org/bots/api#userchatboosts

boosts: List[*ChatBoost*]

The list of boosts added to the chat by the user

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

UserProfilePhotos

class aiogram.types.user_profile_photos.**UserProfilePhotos**(**, total_count: int, photos: List[List[PhotoSize]], **extra_data: Any*)

This object represent a user's profile pictures.

Source: https://core.telegram.org/bots/api#userprofilephotos

total_count: int

Total number of profile pictures the target user has

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

photos: List[List[*PhotoSize*]]

Requested profile pictures (in up to 4 sizes each)

UserShared

class aiogram.types.user_shared.**UserShared**(**, request_id: int, user_id: int, **extra_data: Any*)

This object contains information about the user whose identifier was shared with the bot using a *aiogram.types.keyboard_button_request_user.KeyboardButtonRequestUser* button.

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Source: https://core.telegram.org/bots/api#usershared

request_id: int

Identifier of the request

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_id: int

Identifier of the shared user. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a 64-bit integer or double-precision float type are safe for storing this identifier. The bot may not have access to the user and could be unable to use this identifier, unless the user is already known to the bot by some other means.

UsersShared

class aiogram.types.users_shared.**UsersShared**(**, request_id: int, users: List[SharedUser], user_ids: List[int] | None = None, **extra_data: Any*)

This object contains information about the users whose identifiers were shared with the bot using a *aiogram.types.keyboard_button_request_users.KeyboardButtonRequestUsers* button.

Source: https://core.telegram.org/bots/api#usersshared

request_id: int

Identifier of the request

users: List[*SharedUser*]

Information about users shared with the bot.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_ids: List[int] | None

Identifiers of the shared users. These numbers may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting them. But they have at most 52 significant bits, so 64-bit integers or double-precision float types are safe for storing these identifiers. The bot may not have access to the users and could be unable to use these identifiers, unless the users are already known to the bot by some other means.

Deprecated since version API:7.2: https://core.telegram.org/bots/api-changelog#march-31-2024

Venue

class `aiogram.types.venue.`**`Venue`**(**, location: Location, title: str, address: str, foursquare_id: str | None = None, foursquare_type: str | None = None, google_place_id: str | None = None, google_place_type: str | None = None, **extra_data: Any*)

This object represents a venue.

Source: https://core.telegram.org/bots/api#venue

`location:` ***`Location`***

Venue location. Can't be a live location

`title: str`

Name of the venue

`address: str`

Address of the venue

`foursquare_id: str | None`

Optional. Foursquare identifier of the venue

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`foursquare_type: str | None`

Optional. Foursquare type of the venue. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)

`google_place_id: str | None`

Optional. Google Places identifier of the venue

`google_place_type: str | None`

Optional. Google Places type of the venue. (See supported types.)

Video

class `aiogram.types.video.`**`Video`**(**, file_id: str, file_unique_id: str, width: int, height: int, duration: int, thumbnail: PhotoSize | None = None, file_name: str | None = None, mime_type: str | None = None, file_size: int | None = None, **extra_data: Any*)

This object represents a video file.

Source: https://core.telegram.org/bots/api#video

`file_id: str`

Identifier for this file, which can be used to download or reuse the file

`file_unique_id: str`

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

`width: int`

Video width as defined by the sender

height: int

Video height as defined by the sender

duration: int

Duration of the video in seconds as defined by the sender

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

thumbnail: *PhotoSize* | None

Optional. Video thumbnail

file_name: str | None

Optional. Original filename as defined by the sender

mime_type: str | None

Optional. MIME type of the file as defined by the sender

file_size: int | None

Optional. File size in bytes. It can be bigger than 2^31 and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this value.

VideoChatEnded

class aiogram.types.video_chat_ended.**VideoChatEnded**(*, *duration: int*, ***extra_data: Any*)

This object represents a service message about a video chat ended in the chat.

Source: https://core.telegram.org/bots/api#videochatended

duration: int

Video chat duration in seconds

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

VideoChatParticipantsInvited

class aiogram.types.video_chat_participants_invited.**VideoChatParticipantsInvited**(*, *users: List[User]*, ***extra_data: Any*)

This object represents a service message about new members invited to a video chat.

Source: https://core.telegram.org/bots/api#videochatparticipantsinvited

users: List[*User*]

New members that were invited to the video chat

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

VideoChatScheduled

class aiogram.types.video_chat_scheduled.**VideoChatScheduled**(***, *start_date: _datetime_serializer, return_type=int, when_used=unless - none)]*, ***extra_data: Any*)

This object represents a service message about a video chat scheduled in the chat.

Source: https://core.telegram.org/bots/api#videochatscheduled

start_date: DateTime

Point in time (Unix timestamp) when the video chat is supposed to be started by a chat administrator

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

VideoChatStarted

class aiogram.types.video_chat_started.**VideoChatStarted**(***extra_data: Any*)

This object represents a service message about a video chat started in the chat. Currently holds no information.

Source: https://core.telegram.org/bots/api#videochatstarted

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

VideoNote

class aiogram.types.video_note.**VideoNote**(***, *file_id: str*, *file_unique_id: str*, *length: int*, *duration: int*, *thumbnail: PhotoSize | None = None*, *file_size: int | None = None*, ***extra_data: Any*)

This object represents a video message (available in Telegram apps as of v.4.0).

Source: https://core.telegram.org/bots/api#videonote

file_id: str

Identifier for this file, which can be used to download or reuse the file

file_unique_id: str

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

length: int

Video width and height (diameter of the video message) as defined by the sender

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

duration: int

Duration of the video in seconds as defined by the sender

thumbnail: ***PhotoSize*** **| None**

Optional. Video thumbnail

file_size: int | None

Optional. File size in bytes

Voice

class aiogram.types.voice.**Voice**(**, file_id: str, file_unique_id: str, duration: int, mime_type: str | None = None, file_size: int | None = None, **extra_data: Any*)

This object represents a voice note.

Source: https://core.telegram.org/bots/api#voice

file_id: str

Identifier for this file, which can be used to download or reuse the file

file_unique_id: str

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

duration: int

Duration of the audio in seconds as defined by the sender

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

mime_type: str | None

Optional. MIME type of the file as defined by the sender

file_size: int | None

Optional. File size in bytes. It can be bigger than 2^31 and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this value.

WebAppData

class `aiogram.types.web_app_data.`**`WebAppData`**(*, *data: str*, *button_text: str*, ***extra_data: Any*)

Describes data sent from a Web App to the bot.

Source: https://core.telegram.org/bots/api#webappdata

`data: str`

The data. Be aware that a bad client can send arbitrary data in this field.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`button_text: str`

Text of the *web_app* keyboard button from which the Web App was opened. Be aware that a bad client can send arbitrary data in this field.

WebAppInfo

class `aiogram.types.web_app_info.`**`WebAppInfo`**(*, *url: str*, ***extra_data: Any*)

Describes a Web App.

Source: https://core.telegram.org/bots/api#webappinfo

`url: str`

An HTTPS URL of a Web App to be opened with additional data as specified in Initializing Web Apps

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

WriteAccessAllowed

class `aiogram.types.write_access_allowed.`**`WriteAccessAllowed`**(*, *from_request: bool | None = None*, *web_app_name: str | None = None*, *from_attachment_menu: bool | None = None*, ***extra_data: Any*)

This object represents a service message about a user allowing a bot to write messages after adding it to the attachment menu, launching a Web App from a link, or accepting an explicit request from a Web App sent by the method requestWriteAccess.

Source: https://core.telegram.org/bots/api#writeaccessallowed

`from_request: bool | None`

Optional. True, if the access was granted after the user accepted an explicit request from a Web App sent by the method requestWriteAccess

`web_app_name: str | None`

Optional. Name of the Web App, if the access was granted when the Web App was launched from a link

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

from_attachment_menu: bool | None

Optional. True, if the access was granted when the bot was added to the attachment or side menu

Inline mode

ChosenInlineResult

class aiogram.types.chosen_inline_result.**ChosenInlineResult**(**, result_id: str, from_user: User, query: str, location: Location | None = None, inline_message_id: str | None = None, **extra_data: Any*)

Represents a result of an inline query that was chosen by the user and sent to their chat partner. **Note:** It is necessary to enable inline feedback via @BotFather in order to receive these objects in updates.

Source: https://core.telegram.org/bots/api#choseninlineresult

result_id: str

The unique identifier for the result that was chosen

from_user: *User*

The user that chose the result

query: str

The query that was used to obtain the result

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

location: *Location* | None

Optional. Sender location, only for bots that require user location

inline_message_id: str | None

Optional. Identifier of the sent inline message. Available only if there is an inline keyboard attached to the message. Will be also received in callback queries and can be used to edit the message.

InlineQuery

class aiogram.types.inline_query.**InlineQuery**(**, id: str, from_user: User, query: str, offset: str, chat_type: str | None = None, location: Location | None = None, **extra_data: Any*)

This object represents an incoming inline query. When the user sends an empty query, your bot could return some default or trending results.

Source: https://core.telegram.org/bots/api#inlinequery

id: str
Unique identifier for this query

from_user: *User*
Sender

query: str
Text of the query (up to 256 characters)

offset: str
Offset of the results to be returned, can be controlled by the bot

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

chat_type: str | None
Optional. Type of the chat from which the inline query was sent. Can be either 'sender' for a private chat with the inline query sender, 'private', 'group', 'supergroup', or 'channel'. The chat type should be always known for requests sent from official clients and most third-party clients, unless the request was sent from a secret chat

location: *Location* | None
Optional. Sender location, only for bots that request user location

answer(*results: List[*InlineQueryResultCachedAudio | InlineQueryResultCachedDocument | InlineQueryResultCachedGif | InlineQueryResultCachedMpeg4Gif | InlineQueryResultCachedPhoto | InlineQueryResultCachedSticker | InlineQueryResultCachedVideo | InlineQueryResultCachedVoice | InlineQueryResultArticle | InlineQueryResultAudio | InlineQueryResultContact | InlineQueryResultGame | InlineQueryResultDocument | InlineQueryResultGif | InlineQueryResultLocation | InlineQueryResultMpeg4Gif | InlineQueryResultPhoto | InlineQueryResultVenue | InlineQueryResultVideo | InlineQueryResultVoice*]*, *cache_time: int* | *None = None*, *is_personal: bool* | *None = None*, *next_offset: str* | *None = None*, *button:* InlineQueryResultsButton | *None = None*, *switch_pm_parameter: str* | *None = None*, *switch_pm_text: str* | *None = None*, ***kwargs: Any*) → *AnswerInlineQuery*

Shortcut for method `aiogram.methods.answer_inline_query.AnswerInlineQuery` will automatically fill method attributes:

- `inline_query_id`

Use this method to send answers to an inline query. On success, `True` is returned.

No more than **50** results per query are allowed.

Source: https://core.telegram.org/bots/api#answerinlinequery

Parameters

- **results** – A JSON-serialized array of results for the inline query
- **cache_time** – The maximum amount of time in seconds that the result of the inline query may be cached on the server. Defaults to 300.
- **is_personal** – Pass `True` if results may be cached on the server side only for the user that sent the query. By default, results may be returned to any user who sends the same query.
- **next_offset** – Pass the offset that a client should send in the next query with the same text to receive more results. Pass an empty string if there are no more results or if you don't support pagination. Offset length can't exceed 64 bytes.

- **button** – A JSON-serialized object describing a button to be shown above inline query results
- **switch_pm_parameter** – Deep-linking parameter for the /start message sent to the bot when user presses the switch button. 1-64 characters, only `A-Z`, `a-z`, `0-9`, `_` and `-` are allowed.
- **switch_pm_text** – If passed, clients will display a button with specified text that switches the user to a private chat with the bot and sends the bot a start message with the parameter *switch_pm_parameter*

Returns
instance of method *aiogram.methods.answer_inline_query.AnswerInlineQuery*

InlineQueryResult

`class aiogram.types.inline_query_result.InlineQueryResult(**extra_data: Any)`

This object represents one result of an inline query. Telegram clients currently support results of the following 20 types:

- *aiogram.types.inline_query_result_cached_audio.InlineQueryResultCachedAudio*
- *aiogram.types.inline_query_result_cached_document.InlineQueryResultCachedDocument*
- *aiogram.types.inline_query_result_cached_gif.InlineQueryResultCachedGif*
- *aiogram.types.inline_query_result_cached_mpeg4_gif.InlineQueryResultCachedMpeg4Gif*
- *aiogram.types.inline_query_result_cached_photo.InlineQueryResultCachedPhoto*
- *aiogram.types.inline_query_result_cached_sticker.InlineQueryResultCachedSticker*
- *aiogram.types.inline_query_result_cached_video.InlineQueryResultCachedVideo*
- *aiogram.types.inline_query_result_cached_voice.InlineQueryResultCachedVoice*
- *aiogram.types.inline_query_result_article.InlineQueryResultArticle*
- *aiogram.types.inline_query_result_audio.InlineQueryResultAudio*
- *aiogram.types.inline_query_result_contact.InlineQueryResultContact*
- *aiogram.types.inline_query_result_game.InlineQueryResultGame*
- *aiogram.types.inline_query_result_document.InlineQueryResultDocument*
- *aiogram.types.inline_query_result_gif.InlineQueryResultGif*
- *aiogram.types.inline_query_result_location.InlineQueryResultLocation*
- *aiogram.types.inline_query_result_mpeg4_gif.InlineQueryResultMpeg4Gif*
- *aiogram.types.inline_query_result_photo.InlineQueryResultPhoto*
- *aiogram.types.inline_query_result_venue.InlineQueryResultVenue*
- *aiogram.types.inline_query_result_video.InlineQueryResultVideo*
- *aiogram.types.inline_query_result_voice.InlineQueryResultVoice*

Note: All URLs passed in inline query results will be available to end users and therefore must be assumed to be **public**.

Source: https://core.telegram.org/bots/api#inlinequeryresult

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

InlineQueryResultArticle

class aiogram.types.inline_query_result_article.**InlineQueryResultArticle**(*, *type: Literal[InlineQueryResultType.ARTICLE] = InlineQueryResultType.ARTICLE*, *id: str*, *title: str*, *input_message_content:* InputTextMessageContent | InputLocationMessageContent | InputVenueMessageContent | InputContactMessageContent | InputInvoiceMessageContent, *reply_markup:* InlineKeyboardMarkup *| None = None*, *url: str | None = None*, *hide_url: bool | None = None*, *description: str | None = None*, *thumbnail_url: str | None = None*, *thumbnail_width: int | None = None*, *thumbnail_height: int | None = None*, ***extra_data: Any*)

Represents a link to an article or web page.

Source: https://core.telegram.org/bots/api#inlinequeryresultarticle

type: Literal[InlineQueryResultType.ARTICLE]

Type of the result, must be *article*

id: str

Unique identifier for this result, 1-64 Bytes

title: str

Title of the result

input_message_content: *InputTextMessageContent* **|** *InputLocationMessageContent* **|** *InputVenueMessageContent* **|** *InputContactMessageContent* **|** *InputInvoiceMessageContent*

Content of the message to be sent

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

url: str | None

Optional. URL of the result

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

hide_url: bool | None

Optional. Pass `True` if you don't want the URL to be shown in the message

description: str | None

Optional. Short description of the result

thumbnail_url: str | None

Optional. Url of the thumbnail for the result

thumbnail_width: int | None

Optional. Thumbnail width

thumbnail_height: int | None

Optional. Thumbnail height

InlineQueryResultAudio

class aiogram.types.inline_query_result_audio.**InlineQueryResultAudio**(**, type: ~typing.Literal[InlineQueryResultType.AUDIO] = InlineQueryResultType.AUDIO, id: str, audio_url: str, title: str, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.Mess | None = None, performer: str | None = None, audio_duration: int | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.Inli | None = None, input_message_content: ~aiogram.types.input_text_message_content. | ~aiogram.types.input_location_message_con | ~aiogram.types.input_venue_message_conter | ~aiogram.types.input_contact_message_cont | ~aiogram.types.input_invoice_message_conte | None = None, **extra_data: ~typing.Any*)

Represents a link to an MP3 audio file. By default, this audio file will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the audio.

Source: https://core.telegram.org/bots/api#inlinequeryresultaudio

type: Literal[InlineQueryResultType.AUDIO]

Type of the result, must be *audio*

id: str

Unique identifier for this result, 1-64 bytes

audio_url: str

A valid URL for the audio file

title: str

Title

caption: str | None

Optional. Caption, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the audio caption. See formatting options for more details.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

performer: str | None

Optional. Performer

audio_duration: int | None

Optional. Audio duration in seconds

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the audio

InlineQueryResultCachedAudio

class aiogram.types.inline_query_result_cached_audio.**InlineQueryResultCachedAudio**(**, type: ~typing.Literal[InlineQueryResu = InlineQueryResultType.AUDIO, id: str, audio_file_id: str, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Defa | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.mes | None = None, reply_markup: ~aiogram.types.inline_keybo | None = None, input_message_content: ~aiogram.types.input_text_r | ~aiogram.types.input_locati | ~aiogram.types.input_venue | ~aiogram.types.input_conta | ~aiogram.types.input_invoic | None = None, **extra_data: ~typing.Any*)

Represents a link to an MP3 audio file stored on the Telegram servers. By default, this audio file will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the audio.

Source: https://core.telegram.org/bots/api#inlinequeryresultcachedaudio

type: Literal[InlineQueryResultType.AUDIO]

Type of the result, must be *audio*

id: str

Unique identifier for this result, 1-64 bytes

audio_file_id: str

A valid file identifier for the audio file

caption: str | None

Optional. Caption, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the audio caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the audio

InlineQueryResultCachedDocument

class aiogram.types.inline_query_result_cached_document.**InlineQueryResultCachedDocument**(**,*
type:
~typ-
ing.Literal[InlineQu
=
In-
line-
QueryRe-
sult-
Type.DOCUMENT,
id:
str,
ti-
tle:
str,
doc-
u-
ment_file_id:
str,
de-
scrip-
tion:
str
|
None
=
None,
cap-
tion:
str
|
None
=
None,
parse_mode:
str
|
~aiogram.client.defa
|
None
=
<De-
fault('parse_mode')>
cap-
tion_entities:
~typ-
ing.List[~aiogram.ty
|
None
=
None,
re-
ply_markup:
~aiogram.types.inlin
|
None
=
None,
in-
put_message_conten

Represents a link to a file stored on the Telegram servers. By default, this file will be sent by the user with an optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the file.

Source: https://core.telegram.org/bots/api#inlinequeryresultcacheddocument

type: Literal[InlineQueryResultType.DOCUMENT]

Type of the result, must be *document*

id: str

Unique identifier for this result, 1-64 bytes

title: str

Title for the result

document_file_id: str

A valid file identifier for the file

description: str | None

Optional. Short description of the result

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

caption: str | None

Optional. Caption of the document to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the document caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the file

InlineQueryResultCachedGif

class aiogram.types.inline_query_result_cached_gif.**InlineQueryResultCachedGif**(**, type: ~typing.Literal[InlineQueryResultTyp = InlineQueryResultType.GIF, id: str, gif_file_id: str, title: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_medi reply_markup: ~aiogram.types.inline_keyboard | None = None, input_message_content: ~aiogram.types.input_text_messa | ~aiogram.types.input_location_m | ~aiogram.types.input_venue_mes | ~aiogram.types.input_contact_m | ~aiogram.types.input_invoice_m | None = None, **extra_data: ~typing.Any*)

Represents a link to an animated GIF file stored on the Telegram servers. By default, this animated GIF file will be sent by the user with an optional caption. Alternatively, you can use *input_message_content* to send a message with specified content instead of the animation.

Source: https://core.telegram.org/bots/api#inlinequeryresultcachedgif

type: Literal[InlineQueryResultType.GIF]

Type of the result, must be *gif*

id: str

Unique identifier for this result, 1-64 bytes

gif_file_id: str

A valid file identifier for the GIF file

title: str | None

Optional. Title for the result

caption: str | None

Optional. Caption of the GIF file to be sent, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the GIF animation

InlineQueryResultCachedMpeg4Gif

class aiogram.types.inline_query_result_cached_mpeg4_gif.**InlineQueryResultCachedMpeg4Gif**(*, *type: ~typing.Literal[InlineQ = InlineQueryResultType.MPEG4_GIF id: str, mpeg4_file_id: str, title: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.de | None = <Default('parse_mode' caption_entities: ~typing.List[~aiogram. | None = None, show_caption_abo bool | ~aiogram.client.de | None = <Default('show_captio reply_markup: ~aiogram.types.inl | None =*

Represents a link to a video animation (H.264/MPEG-4 AVC video without sound) stored on the Telegram servers. By default, this animated MPEG-4 file will be sent by the user with an optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the animation.

Source: https://core.telegram.org/bots/api#inlinequeryresultcachedmpeg4gif

type: Literal[InlineQueryResultType.MPEG4_GIF]

Type of the result, must be *mpeg4_gif*

id: str

Unique identifier for this result, 1-64 bytes

mpeg4_file_id: str

A valid file identifier for the MPEG4 file

title: str | None

Optional. Title for the result

caption: str | None

Optional. Caption of the MPEG-4 file to be sent, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the video animation

InlineQueryResultCachedPhoto

class aiogram.types.inline_query_result_cached_photo.**InlineQueryResultCachedPhoto**(**, type: ~typing.Literal[InlineQueryResu = InlineQueryResultType.PHOTO, id: str, photo_file_id: str, title: str | None = None, description: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Defa | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.mes | None = None, show_caption_above_media bool | ~aiogram.client.default.Defa | None = <Default('show_caption_above_ reply_markup: ~aiogram.types.inline_keybo | None = None, input_message_content: ~aiogram.types.input_text_r | ~aiogram.types.input_locati | ~aiogram.types.input_venue | ~aiogram.types.input_conta | ~aiogram.types.input_invoic | None = None, **extra_data: ~typing.Any*)

Represents a link to a photo stored on the Telegram servers. By default, this photo will be sent by the user with an optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the photo.

Source: https://core.telegram.org/bots/api#inlinequeryresultcachedphoto

type: Literal[InlineQueryResultType.PHOTO]

Type of the result, must be *photo*

id: str

Unique identifier for this result, 1-64 bytes

photo_file_id: str

A valid file identifier of the photo

title: str | None

Optional. Title for the result

description: str | None

Optional. Short description of the result

caption: str | None

Optional. Caption of the photo to be sent, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the photo caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the photo

InlineQueryResultCachedSticker

class aiogram.types.inline_query_result_cached_sticker.**InlineQueryResultCachedSticker**(*, *type: Literal[InlineQueryResul = InlineQueryResultType.STICKER*, *id: str*, *sticker_file_id: str*, *reply_markup: InlineKeyboardMarkup | None = None*, *input_message_content: InputTextMessageContent | InputLocationMessageContent | InputVenueMessageContent | InputContactMessageContent | InputInvoiceMessageContent*

Represents a link to a sticker stored on the Telegram servers. By default, this sticker will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the sticker.

Source: https://core.telegram.org/bots/api#inlinequeryresultcachedsticker

type: Literal[InlineQueryResultType.STICKER]

Type of the result, must be *sticker*

id: str

Unique identifier for this result, 1-64 bytes

sticker_file_id: str

A valid file identifier of the sticker

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

reply_markup: ***InlineKeyboardMarkup*** **| None**

Optional. Inline keyboard attached to the message

input_message_content: ***InputTextMessageContent*** **|** ***InputLocationMessageContent*** **|** ***InputVenueMessageContent*** **|** ***InputContactMessageContent*** **|** ***InputInvoiceMessageContent*** **| None**

Optional. Content of the message to be sent instead of the sticker

InlineQueryResultCachedVideo

class aiogram.types.inline_query_result_cached_video.**InlineQueryResultCachedVideo**(**, type: ~typing.Literal[InlineQueryResu = InlineQueryResultType.VIDEO, id: str, video_file_id: str, title: str, description: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Def | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.me | None = None, show_caption_above_medi bool | ~aiogram.client.default.Def | None = <Default('show_caption_above_ reply_markup: ~aiogram.types.inline_keyb | None = None, input_message_content: ~aiogram.types.input_text_ | ~aiogram.types.input_locat | ~aiogram.types.input_venue | ~aiogram.types.input_conta | ~aiogram.types.input_invoi | None = None, **extra_data: ~typing.Any*)

Represents a link to a video file stored on the Telegram servers. By default, this video file will be sent by the user with an optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the video.

Source: https://core.telegram.org/bots/api#inlinequeryresultcachedvideo

type: Literal[InlineQueryResultType.VIDEO]

Type of the result, must be *video*

id: str

Unique identifier for this result, 1-64 bytes

video_file_id: str

A valid file identifier for the video file

title: str

Title for the result

description: str | None

Optional. Short description of the result

caption: str | None

Optional. Caption of the video to be sent, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the video caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Optional. Pass True, if the caption must be shown above the message media

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the video

InlineQueryResultCachedVoice

class aiogram.types.inline_query_result_cached_voice.**InlineQueryResultCachedVoice**(**, type: ~typing.Literal[InlineQueryResu = InlineQueryResultType.VOICE, id: str, voice_file_id: str, title: str, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Defa | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.mes | None = None, reply_markup: ~aiogram.types.inline_keyb | None = None, input_message_content: ~aiogram.types.input_text_r | ~aiogram.types.input_locat | ~aiogram.types.input_venue | ~aiogram.types.input_conta | ~aiogram.types.input_invoic | None = None, **extra_data: ~typing.Any*)

Represents a link to a voice message stored on the Telegram servers. By default, this voice message will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the voice message.

Source: https://core.telegram.org/bots/api#inlinequeryresultcachedvoice

type: **Literal[InlineQueryResultType.VOICE]**

Type of the result, must be *voice*

id: str

Unique identifier for this result, 1-64 bytes

voice_file_id: str

A valid file identifier for the voice message

title: str

Voice message title

caption: str | None

Optional. Caption, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the voice message caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the voice message

InlineQueryResultContact

class aiogram.types.inline_query_result_contact.**InlineQueryResultContact**(*, *type: Literal[InlineQueryResultType.CONTACT = InlineQueryResultType.CONTACT*, *id: str*, *phone_number: str*, *first_name: str*, *last_name: str | None = None*, *vcard: str | None = None*, *reply_markup:* InlineKeyboardMarkup *| None = None*, *input_message_content:* InputTextMessageContent | InputLocationMessageContent | InputVenueMessageContent | InputContactMessageContent | InputInvoiceMessageContent *| None = None*, *thumbnail_url: str | None = None*, *thumbnail_width: int | None = None*, *thumbnail_height: int | None = None*, ***extra_data: Any*)

Represents a contact with a phone number. By default, this contact will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the contact.

Source: https://core.telegram.org/bots/api#inlinequeryresultcontact

type: Literal[InlineQueryResultType.CONTACT]

Type of the result, must be *contact*

id: str

Unique identifier for this result, 1-64 Bytes

phone_number: str

Contact’s phone number

first_name: str

Contact’s first name

last_name: str | None

Optional. Contact’s last name

vcard: str | None

Optional. Additional data about the contact in the form of a vCard, 0-2048 bytes

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the contact

thumbnail_url: str | None

Optional. Url of the thumbnail for the result

thumbnail_width: int | None

Optional. Thumbnail width

thumbnail_height: int | None

Optional. Thumbnail height

InlineQueryResultDocument

class aiogram.types.inline_query_result_document.**InlineQueryResultDocument**(**, type: ~typing.Literal[InlineQueryResultType.D = InlineQueryResultType.DOCUMENT, id: str, title: str, document_url: str, mime_type: str, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_ent | None = None, description: str | None = None, reply_markup: ~aiogram.types.inline_keyboard_mar | None = None, input_message_content: ~aiogram.types.input_text_message_ | ~aiogram.types.input_location_messa | ~aiogram.types.input_venue_message | ~aiogram.types.input_contact_messa | ~aiogram.types.input_invoice_messa | None = None, thumbnail_url: str | None = None, thumbnail_width: int | None = None, thumbnail_height: int | None = None, **extra_data: ~typing.Any*)

Represents a link to a file. By default, this file will be sent by the user with an optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the file. Currently, only **.PDF** and **.ZIP** files can be sent using this method.

Source: https://core.telegram.org/bots/api#inlinequeryresultdocument

type: Literal[InlineQueryResultType.DOCUMENT]

Type of the result, must be *document*

id: str

Unique identifier for this result, 1-64 bytes

title: str

Title for the result

document_url: str

A valid URL for the file

mime_type: str

MIME type of the content of the file, either 'application/pdf' or 'application/zip'

caption: str | None

Optional. Caption of the document to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the document caption. See formatting options for more details.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

description: str | None

Optional. Short description of the result

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the file

thumbnail_url: str | None

Optional. URL of the thumbnail (JPEG only) for the file

thumbnail_width: int | None

Optional. Thumbnail width

thumbnail_height: int | None

Optional. Thumbnail height

InlineQueryResultGame

class aiogram.types.inline_query_result_game.**InlineQueryResultGame**(**, *type: Literal[InlineQueryResultType.GAME] = InlineQueryResultType.GAME*, *id: str*, *game_short_name: str*, *reply_markup: InlineKeyboardMarkup | None = None*, ***extra_data: Any*)

Represents a Game.

Source: https://core.telegram.org/bots/api#inlinequeryresultgame

type: Literal[InlineQueryResultType.GAME]

Type of the result, must be *game*

id: str

Unique identifier for this result, 1-64 bytes

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

game_short_name: str

Short name of the game

reply_markup: ***InlineKeyboardMarkup*** **| None**

Optional. Inline keyboard attached to the message

InlineQueryResultGif

class aiogram.types.inline_query_result_gif.**InlineQueryResultGif**(**, type: ~typing.Literal[InlineQueryResultType.GIF] = InlineQueryResultType.GIF, id: str, gif_url: str, thumbnail_url: str, gif_width: int | None = None, gif_height: int | None = None, gif_duration: int | None = None, thumbnail_mime_type: str | None = None, title: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageE | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKe | None = None, input_message_content: ~aiogram.types.input_text_message_content.Input | ~aiogram.types.input_location_message_content.I | ~aiogram.types.input_venue_message_content.Inp | ~aiogram.types.input_contact_message_content.In | ~aiogram.types.input_invoice_message_content.In | None = None, **extra_data: ~typing.Any*)

Represents a link to an animated GIF file. By default, this animated GIF file will be sent by the user with optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the animation.

Source: https://core.telegram.org/bots/api#inlinequeryresultgif

type: Literal[InlineQueryResultType.GIF]

Type of the result, must be *gif*

id: str

Unique identifier for this result, 1-64 bytes

gif_url: str

A valid URL for the GIF file. File size must not exceed 1MB

thumbnail_url: str

URL of the static (JPEG or GIF) or animated (MPEG4) thumbnail for the result

`gif_width: int | None`

Optional. Width of the GIF

`gif_height: int | None`

Optional. Height of the GIF

`gif_duration: int | None`

Optional. Duration of the GIF in seconds

`thumbnail_mime_type: str | None`

Optional. MIME type of the thumbnail, must be one of 'image/jpeg', 'image/gif', or 'video/mp4'. Defaults to 'image/jpeg'

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`title: str | None`

Optional. Title for the result

`caption: str | None`

Optional. Caption of the GIF file to be sent, 0-1024 characters after entities parsing

`parse_mode: str | Default | None`

Optional. Mode for parsing entities in the caption. See formatting options for more details.

`caption_entities: List[MessageEntity] | None`

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

`show_caption_above_media: bool | Default | None`

Optional. Pass `True`, if the caption must be shown above the message media

`reply_markup: InlineKeyboardMarkup | None`

Optional. Inline keyboard attached to the message

`input_message_content: InputTextMessageContent | InputLocationMessageContent | InputVenueMessageContent | InputContactMessageContent | InputInvoiceMessageContent | None`

Optional. Content of the message to be sent instead of the GIF animation

InlineQueryResultLocation

`class aiogram.types.inline_query_result_location.InlineQueryResultLocation`(**, type: Literal[InlineQueryResultType.LOCATI = InlineQueryResultType.LOCATION, id: str, latitude: float, longitude: float, title: str, horizontal_accuracy: float | None = None, live_period: int | None = None, heading: int | None = None, proximity_alert_radius: int | None = None, reply_markup:* InlineKeyboardMarkup | *None = None, input_message_content:* InputTextMessageContent | InputLocationMessageContent | InputVenueMessageContent | InputContactMessageContent | InputInvoiceMessageContent | *None = None, thumbnail_url: str | None = None, thumbnail_width: int | None = None, thumbnail_height: int | None = None, **extra_data: Any*)

Represents a location on a map. By default, the location will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the location.

Source: https://core.telegram.org/bots/api#inlinequeryresultlocation

`type: Literal[InlineQueryResultType.LOCATION]`

Type of the result, must be *location*

`id: str`

Unique identifier for this result, 1-64 Bytes

`latitude: float`

Location latitude in degrees

longitude: float

Location longitude in degrees

title: str

Location title

horizontal_accuracy: float | None

Optional. The radius of uncertainty for the location, measured in meters; 0-1500

live_period: int | None

Optional. Period in seconds during which the location can be updated, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

heading: int | None

Optional. For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.

proximity_alert_radius: int | None

Optional. For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the location

thumbnail_url: str | None

Optional. Url of the thumbnail for the result

thumbnail_width: int | None

Optional. Thumbnail width

thumbnail_height: int | None

Optional. Thumbnail height

InlineQueryResultMpeg4Gif

`class aiogram.types.inline_query_result_mpeg4_gif.InlineQueryResultMpeg4Gif`(**, type: ~typing.Literal[InlineQueryResultType.I = InlineQueryResultType.MPEG4_GIF, id: str, mpeg4_url: str, thumbnail_url: str, mpeg4_width: int | None = None, mpeg4_height: int | None = None, mpeg4_duration: int | None = None, thumbnail_mime_type: str | None = None, title: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_e | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media') reply_markup: ~aiogram.types.inline_keyboard_m | None = None, input_message_content: ~aiogram.types.input_text_message | ~aiogram.types.input_location_mes | ~aiogram.types.input_venue_messa | ~aiogram.types.input_contact_mess | ~aiogram.types.input_invoice_mess | None = None, **extra_data: ~typing.Any*)

Represents a link to a video animation (H.264/MPEG-4 AVC video without sound). By default, this animated MPEG-4 file will be sent by the user with optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the animation.

Source: https://core.telegram.org/bots/api#inlinequeryresultmpeg4gif

type: Literal[InlineQueryResultType.MPEG4_GIF]

Type of the result, must be *mpeg4_gif*

id: str

Unique identifier for this result, 1-64 bytes

mpeg4_url: str

A valid URL for the MPEG4 file. File size must not exceed 1MB

thumbnail_url: str

URL of the static (JPEG or GIF) or animated (MPEG4) thumbnail for the result

mpeg4_width: int | None

Optional. Video width

mpeg4_height: int | None

Optional. Video height

mpeg4_duration: int | None

Optional. Video duration in seconds

thumbnail_mime_type: str | None

Optional. MIME type of the thumbnail, must be one of 'image/jpeg', 'image/gif', or 'video/mp4'. Defaults to 'image/jpeg'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

title: str | None

Optional. Title for the result

caption: str | None

Optional. Caption of the MPEG-4 file to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the video animation

InlineQueryResultPhoto

class aiogram.types.inline_query_result_photo.**InlineQueryResultPhoto**(**, type: ~typing.Literal[InlineQueryResultType.PHOTO] = InlineQueryResultType.PHOTO, id: str, photo_url: str, thumbnail_url: str, photo_width: int | None = None, photo_height: int | None = None, title: str | None = None, description: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.Mess | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, reply_markup: ~aiogram.types.inline_keyboard_markup.Inl | None = None, input_message_content: ~aiogram.types.input_text_message_content. | ~aiogram.types.input_location_message_con | ~aiogram.types.input_venue_message_conte | ~aiogram.types.input_contact_message_cont | ~aiogram.types.input_invoice_message_cont | None = None, **extra_data: ~typing.Any*)

Represents a link to a photo. By default, this photo will be sent by the user with optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the photo.

Source: https://core.telegram.org/bots/api#inlinequeryresultphoto

type: Literal[InlineQueryResultType.PHOTO]

Type of the result, must be *photo*

id: str

Unique identifier for this result, 1-64 bytes

photo_url: str

A valid URL of the photo. Photo must be in **JPEG** format. Photo size must not exceed 5MB

thumbnail_url: str

URL of the thumbnail for the photo

photo_width: int | None

Optional. Width of the photo

photo_height: int | None

Optional. Height of the photo

title: str | None

Optional. Title for the result

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

description: str | None

Optional. Short description of the result

caption: str | None

Optional. Caption of the photo to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Optional. Mode for parsing entities in the photo caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Optional. Pass `True`, if the caption must be shown above the message media

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the photo

InlineQueryResultVenue

class aiogram.types.inline_query_result_venue.**InlineQueryResultVenue**(**, type: Literal[InlineQueryResultType.VENUE] = InlineQueryResultType.VENUE, id: str, latitude: float, longitude: float, title: str, address: str, foursquare_id: str | None = None, foursquare_type: str | None = None, google_place_id: str | None = None, google_place_type: str | None = None, reply_markup: InlineKeyboardMarkup | None = None, input_message_content: InputTextMessageContent | InputLocationMessageContent | InputVenueMessageContent | InputContactMessageContent | InputInvoiceMessageContent | None = None, thumbnail_url: str | None = None, thumbnail_width: int | None = None, thumbnail_height: int | None = None, **extra_data: Any*)

Represents a venue. By default, the venue will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the venue.

Source: https://core.telegram.org/bots/api#inlinequeryresultvenue

type: Literal[InlineQueryResultType.VENUE]

Type of the result, must be *venue*

id: str

Unique identifier for this result, 1-64 Bytes

latitude: float

Latitude of the venue location in degrees

longitude: float

Longitude of the venue location in degrees

title: str

Title of the venue

address: str

Address of the venue

foursquare_id: str | None

Optional. Foursquare identifier of the venue if known

foursquare_type: str | None

Optional. Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

google_place_id: str | None

Optional. Google Places identifier of the venue

google_place_type: str | None

Optional. Google Places type of the venue. (See supported types.)

reply_markup: ***InlineKeyboardMarkup*** **| None**

Optional. Inline keyboard attached to the message

input_message_content: ***InputTextMessageContent*** **|** ***InputLocationMessageContent*** **|** ***InputVenueMessageContent*** **|** ***InputContactMessageContent*** **|** ***InputInvoiceMessageContent*** **| None**

Optional. Content of the message to be sent instead of the venue

thumbnail_url: str | None

Optional. Url of the thumbnail for the result

thumbnail_width: int | None

Optional. Thumbnail width

thumbnail_height: int | None

Optional. Thumbnail height

InlineQueryResultVideo

class aiogram.types.inline_query_result_video.**InlineQueryResultVideo**(**, type: ~typing.Literal[InlineQueryResultType.VIDEO] = InlineQueryResultType.VIDEO, id: str, video_url: str, mime_type: str, thumbnail_url: str, title: str, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.Mess | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, video_width: int | None = None, video_height: int | None = None, video_duration: int | None = None, description: str | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.Inli | None = None, input_message_content: ~aiogram.types.input_text_message_content. | ~aiogram.types.input_location_message_con | ~aiogram.types.input_venue_message_conter | ~aiogram.types.input_contact_message_cont | ~aiogram.types.input_invoice_message_conte | None = None, **extra_data: ~typing.Any*)

Represents a link to a page containing an embedded video player or a video file. By default, this video file will be sent by the user with an optional caption. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the video.

> If an InlineQueryResultVideo message contains an embedded video (e.g., YouTube), you **must** replace its content using *input_message_content*.

Source: https://core.telegram.org/bots/api#inlinequeryresultvideo

`type: Literal[InlineQueryResultType.VIDEO]`

Type of the result, must be *video*

id: str
Unique identifier for this result, 1-64 bytes

video_url: str
A valid URL for the embedded video player or video file

mime_type: str
MIME type of the content of the video URL, 'text/html' or 'video/mp4'

thumbnail_url: str
URL of the thumbnail (JPEG only) for the video

title: str
Title for the result

caption: str | None
Optional. Caption of the video to be sent, 0-1024 characters after entities parsing

parse_mode: str | Default | None
Optional. Mode for parsing entities in the video caption. See formatting options for more details.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

caption_entities: List[*MessageEntity*] | None
Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None
Optional. Pass `True`, if the caption must be shown above the message media

video_width: int | None
Optional. Video width

video_height: int | None
Optional. Video height

video_duration: int | None
Optional. Video duration in seconds

description: str | None
Optional. Short description of the result

reply_markup: *InlineKeyboardMarkup* | None
Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None
Optional. Content of the message to be sent instead of the video. This field is **required** if InlineQueryResultVideo is used to send an HTML-page as a result (e.g., a YouTube video).

InlineQueryResultVoice

class aiogram.types.inline_query_result_voice.**InlineQueryResultVoice**(**, type: ~typing.Literal[InlineQueryResultType.VOICE] = InlineQueryResultType.VOICE, id: str, voice_url: str, title: str, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.Mess | None = None, voice_duration: int | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.Inli | None = None, input_message_content: ~aiogram.types.input_text_message_content. | ~aiogram.types.input_location_message_con | ~aiogram.types.input_venue_message_conte | ~aiogram.types.input_contact_message_cont | ~aiogram.types.input_invoice_message_conte | None = None, **extra_data: ~typing.Any*)

Represents a link to a voice recording in an .OGG container encoded with OPUS. By default, this voice recording will be sent by the user. Alternatively, you can use *input_message_content* to send a message with the specified content instead of the the voice message.

Source: https://core.telegram.org/bots/api#inlinequeryresultvoice

type: Literal[InlineQueryResultType.VOICE]

Type of the result, must be *voice*

id: str

Unique identifier for this result, 1-64 bytes

voice_url: str

A valid URL for the voice recording

title: str

Recording title

caption: str | None

Optional. Caption, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Optional. Mode for parsing entities in the voice message caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in the caption, which can be specified instead of *parse_mode*

voice_duration: int | None

Optional. Recording duration in seconds

reply_markup: *InlineKeyboardMarkup* | None

Optional. Inline keyboard attached to the message

input_message_content: *InputTextMessageContent* | *InputLocationMessageContent* | *InputVenueMessageContent* | *InputContactMessageContent* | *InputInvoiceMessageContent* | None

Optional. Content of the message to be sent instead of the voice recording

InlineQueryResultsButton

class aiogram.types.inline_query_results_button.**InlineQueryResultsButton**(**, text: str, web_app: WebAppInfo | None = None, start_parameter: str | None = None, **extra_data: Any*)

This object represents a button to be shown above inline query results. You **must** use exactly one of the optional fields.

Source: https://core.telegram.org/bots/api#inlinequeryresultsbutton

text: str

Label text on the button

web_app: *WebAppInfo* | None

Optional. Description of the Web App that will be launched when the user presses the button. The Web App will be able to switch back to the inline mode using the method switchInlineQuery inside the Web App.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

start_parameter: str | None

Optional. Deep-linking parameter for the /start message sent to the bot when a user presses the button. 1-64 characters, only `A-Z`, `a-z`, `0-9`, `_` and `-` are allowed.

InputContactMessageContent

class aiogram.types.input_contact_message_content.**InputContactMessageContent**(**, phone_number: str, first_name: str, last_name: str | None = None, vcard: str | None = None, **extra_data: Any*)

Represents the content of a contact message to be sent as the result of an inline query.

Source: https://core.telegram.org/bots/api#inputcontactmessagecontent

phone_number: str

Contact's phone number

first_name: str

Contact's first name

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

last_name: str | None

Optional. Contact's last name

vcard: str | None

Optional. Additional data about the contact in the form of a vCard, 0-2048 bytes

InputInvoiceMessageContent

class `aiogram.types.input_invoice_message_content.`**`InputInvoiceMessageContent`**(**, title: str, description: str, payload: str, currency: str, prices: List[*LabeledPrice*], provider_token: str | None = None, max_tip_amount: int | None = None, suggested_tip_amounts: List[int] | None = None, provider_data: str | None = None, photo_url: str | None = None, photo_size: int | None = None, photo_width: int | None = None, photo_height: int | None = None, need_name: bool | None = None, need_phone_number: bool | None = None, need_email: bool | None = None, need_shipping_address: bool | None = None, send_phone_number_to_provider: bool | None = None, send_email_to_provider: bool | None = None, is_flexible: bool | None = None, **extra_data: Any*)

Represents the content of an invoice message to be sent as the result of an inline query.

Source: https://core.telegram.org/bots/api#inputinvoicemessagecontent

`title: str`

Product name, 1-32 characters

description: str

Product description, 1-255 characters

payload: str

Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.

currency: str

Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.

prices: List[*LabeledPrice*]

Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.

provider_token: str | None

Optional. Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.

max_tip_amount: int | None

Optional. The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.

suggested_tip_amounts: List[int] | None

Optional. A JSON-serialized array of suggested amounts of tip in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.

provider_data: str | None

Optional. A JSON-serialized object for data about the invoice, which will be shared with the payment provider. A detailed description of the required fields should be provided by the payment provider.

photo_url: str | None

Optional. URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

photo_size: int | None

Optional. Photo size in bytes

photo_width: int | None

Optional. Photo width

photo_height: int | None

Optional. Photo height

need_name: bool | None

Optional. Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.

need_phone_number: bool | None

Optional. Pass True if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.

need_email: bool | None

Optional. Pass True if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.

need_shipping_address: bool | None

Optional. Pass True if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.

send_phone_number_to_provider: bool | None

Optional. Pass True if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.

send_email_to_provider: bool | None

Optional. Pass True if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.

is_flexible: bool | None

Optional. Pass True if the final price depends on the shipping method. Ignored for payments in Telegram Stars.

InputLocationMessageContent

class aiogram.types.input_location_message_content.**InputLocationMessageContent**(*, *latitude: float*, *longitude: float*, *horizontal_accuracy: float | None = None*, *live_period: int | None = None*, *heading: int | None = None*, *proximity_alert_radius: int | None = None*, ***extra_data: Any*)

Represents the content of a location message to be sent as the result of an inline query.

Source: https://core.telegram.org/bots/api#inputlocationmessagecontent

latitude: float

Latitude of the location in degrees

longitude: float

Longitude of the location in degrees

horizontal_accuracy: float | None

Optional. The radius of uncertainty for the location, measured in meters; 0-1500

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

live_period: int | None

Optional. Period in seconds during which the location can be updated, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.

heading: int | None

Optional. For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.

proximity_alert_radius: int | None

Optional. For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.

InputMessageContent

class aiogram.types.input_message_content.**InputMessageContent**(***extra_data: Any*)

This object represents the content of a message to be sent as a result of an inline query. Telegram clients currently support the following 5 types:

- *aiogram.types.input_text_message_content.InputTextMessageContent*
- *aiogram.types.input_location_message_content.InputLocationMessageContent*
- *aiogram.types.input_venue_message_content.InputVenueMessageContent*
- *aiogram.types.input_contact_message_content.InputContactMessageContent*
- *aiogram.types.input_invoice_message_content.InputInvoiceMessageContent*

Source: https://core.telegram.org/bots/api#inputmessagecontent

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

InputTextMessageContent

class aiogram.types.input_text_message_content.**InputTextMessageContent**(**, message_text: str, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, entities: ~typing.List[~aiogram.types.message_entity.M | None = None, link_preview_options: ~aiogram.types.link_preview_options.Link | ~aiogram.client.default.Default | None = <Default('link_preview')>, disable_web_page_preview: bool | None = None, **extra_data: ~typing.Any*)

Represents the content of a text message to be sent as the result of an inline query.

Source: https://core.telegram.org/bots/api#inputtextmessagecontent

message_text: str

Text of the message to be sent, 1-4096 characters

parse_mode: str | Default | None

Optional. Mode for parsing entities in the message text. See formatting options for more details.

entities: List[*MessageEntity*] | None

Optional. List of special entities that appear in message text, which can be specified instead of *parse_mode*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

link_preview_options: *LinkPreviewOptions* | Default | None

Optional. Link preview generation options for the message

disable_web_page_preview: bool | None

Optional. Disables link previews for links in the sent message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

InputVenueMessageContent

class aiogram.types.input_venue_message_content.**InputVenueMessageContent**(*, *latitude: float*, *longitude: float*, *title: str*, *address: str*, *foursquare_id: str | None = None*, *foursquare_type: str | None = None*, *google_place_id: str | None = None*, *google_place_type: str | None = None*, ***extra_data: Any*)

Represents the content of a venue message to be sent as the result of an inline query.

Source: https://core.telegram.org/bots/api#inputvenuemessagecontent

latitude: float

Latitude of the venue in degrees

longitude: float

Longitude of the venue in degrees

title: str

Name of the venue

address: str

Address of the venue

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

foursquare_id: str | None

Optional. Foursquare identifier of the venue, if known

foursquare_type: str | None

Optional. Foursquare type of the venue, if known. (For example, 'arts_entertainment/default', 'arts_entertainment/aquarium' or 'food/icecream'.)

google_place_id: str | None

Optional. Google Places identifier of the venue

google_place_type: str | None

Optional. Google Places type of the venue. (See supported types.)

SentWebAppMessage

class `aiogram.types.sent_web_app_message.`**`SentWebAppMessage`**(**, inline_message_id: str | None = None, **extra_data: Any*)

Describes an inline message sent by a Web App on behalf of a user.

Source: https://core.telegram.org/bots/api#sentwebappmessage

`inline_message_id: str | None`

Optional. Identifier of the sent inline message. Available only if there is an inline keyboard attached to the message.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Stickers

InputSticker

class `aiogram.types.input_sticker.`**`InputSticker`**(**, sticker: InputFile | str, format: str, emoji_list: List[str], mask_position: MaskPosition | None = None, keywords: List[str] | None = None, **extra_data: Any*)

This object describes a sticker to be added to a sticker set.

Source: https://core.telegram.org/bots/api#inputsticker

`sticker:` ***`InputFile`*** **`| str`**

The added sticker. Pass a *file_id* as a String to send a file that already exists on the Telegram servers, pass an HTTP URL as a String for Telegram to get a file from the Internet, upload a new one using multipart/form-data, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. Animated and video stickers can't be uploaded via HTTP URL. *More information on Sending Files »*

`format: str`

Format of the added sticker, must be one of 'static' for a **.WEBP** or **.PNG** image, 'animated' for a **.TGS** animation, 'video' for a **WEBM** video

`emoji_list: List[str]`

List of 1-20 emoji associated with the sticker

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`mask_position:` ***`MaskPosition`*** **`| None`**

Optional. Position where the mask should be placed on faces. For 'mask' stickers only.

`keywords: List[str] | None`

Optional. List of 0-20 search keywords for the sticker with total length of up to 64 characters. For 'regular' and 'custom_emoji' stickers only.

MaskPosition

class aiogram.types.mask_position.**MaskPosition**(**, point: str, x_shift: float, y_shift: float, scale: float, **extra_data: Any*)

This object describes the position on faces where a mask should be placed by default.

Source: https://core.telegram.org/bots/api#maskposition

point: str

The part of the face relative to which the mask should be placed. One of 'forehead', 'eyes', 'mouth', or 'chin'.

x_shift: float

Shift by X-axis measured in widths of the mask scaled to the face size, from left to right. For example, choosing -1.0 will place mask just to the left of the default mask position.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

y_shift: float

Shift by Y-axis measured in heights of the mask scaled to the face size, from top to bottom. For example, 1.0 will place the mask just below the default mask position.

scale: float

Mask scaling coefficient. For example, 2.0 means double size.

Sticker

class aiogram.types.sticker.**Sticker**(**, file_id: str, file_unique_id: str, type: str, width: int, height: int, is_animated: bool, is_video: bool, thumbnail: PhotoSize | None = None, emoji: str | None = None, set_name: str | None = None, premium_animation: File | None = None, mask_position: MaskPosition | None = None, custom_emoji_id: str | None = None, needs_repainting: bool | None = None, file_size: int | None = None, **extra_data: Any*)

This object represents a sticker.

Source: https://core.telegram.org/bots/api#sticker

file_id: str

Identifier for this file, which can be used to download or reuse the file

file_unique_id: str

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

type: str

Type of the sticker, currently one of 'regular', 'mask', 'custom_emoji'. The type of the sticker is independent from its format, which is determined by the fields *is_animated* and *is_video*.

width: int

Sticker width

height: int

Sticker height

is_animated: bool

`True`, if the sticker is animated

is_video: bool

`True`, if the sticker is a video sticker

thumbnail: *PhotoSize* | None

Optional. Sticker thumbnail in the .WEBP or .JPG format

emoji: str | None

Optional. Emoji associated with the sticker

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

set_name: str | None

Optional. Name of the sticker set to which the sticker belongs

premium_animation: *File* | None

Optional. For premium regular stickers, premium animation for the sticker

mask_position: *MaskPosition* | None

Optional. For mask stickers, the position where the mask should be placed

custom_emoji_id: str | None

Optional. For custom emoji stickers, unique identifier of the custom emoji

needs_repainting: bool | None

Optional. `True`, if the sticker must be repainted to a text color in messages, the color of the Telegram Premium badge in emoji status, white color on chat photos, or another appropriate color in other places

file_size: int | None

Optional. File size in bytes

set_position_in_set(*position: int*, ***kwargs: Any*) → *SetStickerPositionInSet*

Shortcut for method *`aiogram.methods.set_sticker_position_in_set.SetStickerPositionInSet`* will automatically fill method attributes:

- `sticker`

Use this method to move a sticker in a set created by the bot to a specific position. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setstickerpositioninset

Parameters

position – New sticker position in the set, zero-based

Returns

instance of method *`aiogram.methods.set_sticker_position_in_set.SetStickerPositionInSet`*

delete_from_set(***kwargs: Any*) → *DeleteStickerFromSet*

Shortcut for method `aiogram.methods.delete_sticker_from_set.DeleteStickerFromSet` will automatically fill method attributes:

- `sticker`

Use this method to delete a sticker from a set created by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletestickerfromset

Returns
instance of method `aiogram.methods.delete_sticker_from_set.DeleteStickerFromSet`

StickerSet

class aiogram.types.sticker_set.**StickerSet**(**, name: str, title: str, sticker_type: str, stickers: List[Sticker], thumbnail: PhotoSize | None = None, is_animated: bool | None = None, is_video: bool | None = None, **extra_data: Any*)

This object represents a sticker set.

Source: https://core.telegram.org/bots/api#stickerset

name: str

Sticker set name

title: str

Sticker set title

sticker_type: str

Type of stickers in the set, currently one of 'regular', 'mask', 'custom_emoji'

stickers: List[*Sticker*]

List of all set stickers

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

thumbnail: *PhotoSize* | None

Optional. Sticker set thumbnail in the .WEBP, .TGS, or .WEBM format

is_animated: bool | None

`True`, if the sticker set contains animated stickers

Deprecated since version API:7.2: https://core.telegram.org/bots/api-changelog#march-31-2024

is_video: bool | None

`True`, if the sticker set contains video stickers

Deprecated since version API:7.2: https://core.telegram.org/bots/api-changelog#march-31-2024

Payments

Invoice

class `aiogram.types.invoice.`**`Invoice`**(*, *title: str*, *description: str*, *start_parameter: str*, *currency: str*, *total_amount: int*, ***extra_data: Any*)

This object contains basic information about an invoice.

Source: https://core.telegram.org/bots/api#invoice

`title: str`

Product name

`description: str`

Product description

`start_parameter: str`

Unique bot deep-linking parameter that can be used to generate this invoice

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`currency: str`

Three-letter ISO 4217 currency code, or 'XTR' for payments in Telegram Stars

`total_amount: int`

Total price in the *smallest units* of the currency (integer, **not** float/double). For example, for a price of `US$ 1.45` pass `amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies).

LabeledPrice

class `aiogram.types.labeled_price.`**`LabeledPrice`**(*, *label: str*, *amount: int*, ***extra_data: Any*)

This object represents a portion of the price for goods or services.

Source: https://core.telegram.org/bots/api#labeledprice

`label: str`

Portion label

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`amount: int`

Price of the product in the *smallest units* of the currency (integer, **not** float/double). For example, for a price of `US$ 1.45` pass `amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies).

OrderInfo

class aiogram.types.order_info.**OrderInfo**(**, name: str | None = None, phone_number: str | None = None, email: str | None = None, shipping_address: ShippingAddress | None = None, **extra_data: Any*)

This object represents information about an order.

Source: https://core.telegram.org/bots/api#orderinfo

name: str | None

Optional. User name

phone_number: str | None

Optional. User's phone number

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

email: str | None

Optional. User email

shipping_address: *ShippingAddress* | None

Optional. User shipping address

PaidMediaPurchased

class aiogram.types.paid_media_purchased.**PaidMediaPurchased**(**, from_user: User, paid_media_payload: str, **extra_data: Any*)

This object contains information about a paid media purchase.

Source: https://core.telegram.org/bots/api#paidmediapurchased

from_user: *User*

User who purchased the media

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

paid_media_payload: str

Bot-specified paid media payload

PreCheckoutQuery

class aiogram.types.pre_checkout_query.**PreCheckoutQuery**(**, id: str, from_user: User, currency: str, total_amount: int, invoice_payload: str, shipping_option_id: str | None = None, order_info: OrderInfo | None = None, **extra_data: Any*)

This object contains information about an incoming pre-checkout query.

Source: https://core.telegram.org/bots/api#precheckoutquery

id: str

Unique query identifier

from_user: *User*

User who sent the query

currency: str

Three-letter ISO 4217 currency code, or 'XTR' for payments in Telegram Stars

total_amount: int

Total price in the *smallest units* of the currency (integer, **not** float/double). For example, for a price of `US$ 1.45` pass `amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies).

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

invoice_payload: str

Bot-specified invoice payload

shipping_option_id: str | None

Optional. Identifier of the shipping option chosen by the user

order_info: *OrderInfo* | None

Optional. Order information provided by the user

answer(*ok: bool, error_message: str | None = None, **kwargs: Any*) → *AnswerPreCheckoutQuery*

Shortcut for method *aiogram.methods.answer_pre_checkout_query.AnswerPreCheckoutQuery* will automatically fill method attributes:

- `pre_checkout_query_id`

Once the user has confirmed their payment and shipping details, the Bot API sends the final confirmation in the form of an *aiogram.types.update.Update* with the field *pre_checkout_query*. Use this method to respond to such pre-checkout queries. On success, `True` is returned. **Note:** The Bot API must receive an answer within 10 seconds after the pre-checkout query was sent.

Source: https://core.telegram.org/bots/api#answerprecheckoutquery

Parameters

- **ok** – Specify `True` if everything is alright (goods are available, etc.) and the bot is ready to proceed with the order. Use `False` if there are any problems.

- **error_message** – Required if *ok* is `False`. Error message in human readable form that explains the reason for failure to proceed with the checkout (e.g. "Sorry, somebody just bought the last of our amazing black T-shirts while you were busy filling out your payment details. Please choose a different color or garment!"). Telegram will display this message to the user.

Returns
instance of method *aiogram.methods.answer_pre_checkout_query.AnswerPreCheckoutQuery*

RefundedPayment

class aiogram.types.refunded_payment.**RefundedPayment**(*, *currency: Literal['XTR'] = 'XTR'*, *total_amount: int*, *invoice_payload: str*, *telegram_payment_charge_id: str*, *provider_payment_charge_id: str | None = None*, ***extra_data: Any*)

This object contains basic information about a refunded payment.

Source: https://core.telegram.org/bots/api#refundedpayment

currency: Literal['XTR']

Three-letter ISO 4217 currency code, or 'XTR' for payments in Telegram Stars. Currently, always 'XTR'

total_amount: int

Total refunded price in the *smallest units* of the currency (integer, **not** float/double). For example, for a price of `US$ 1.45`, `total_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies).

invoice_payload: str

Bot-specified invoice payload

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

telegram_payment_charge_id: str

Telegram payment identifier

provider_payment_charge_id: str | None

Optional. Provider payment identifier

RevenueWithdrawalState

class aiogram.types.revenue_withdrawal_state.**RevenueWithdrawalState**(***extra_data: Any*)

This object describes the state of a revenue withdrawal operation. Currently, it can be one of

- *aiogram.types.revenue_withdrawal_state_pending.RevenueWithdrawalStatePending*
- *aiogram.types.revenue_withdrawal_state_succeeded.RevenueWithdrawalStateSucceeded*
- *aiogram.types.revenue_withdrawal_state_failed.RevenueWithdrawalStateFailed*

Source: https://core.telegram.org/bots/api#revenuewithdrawalstate

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

RevenueWithdrawalStateFailed

class aiogram.types.revenue_withdrawal_state_failed.**RevenueWithdrawalStateFailed**(*, *type: Literal[RevenueWithdrawalState = RevenueWithdrawalStateType.FAILED*, ***extra_data: Any*)

The withdrawal failed and the transaction was refunded.

Source: https://core.telegram.org/bots/api#revenuewithdrawalstatefailed

type: Literal[RevenueWithdrawalStateType.FAILED]

Type of the state, always 'failed'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

RevenueWithdrawalStatePending

class aiogram.types.revenue_withdrawal_state_pending.**RevenueWithdrawalStatePending**(*, *type: Literal[RevenueWithdrawalSt = RevenueWithdrawalStateType.PENDING*, ***extra_data: Any*)

The withdrawal is in progress.

Source: https://core.telegram.org/bots/api#revenuewithdrawalstatepending

type: Literal[RevenueWithdrawalStateType.PENDING]

Type of the state, always 'pending'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

RevenueWithdrawalStateSucceeded

class aiogram.types.revenue_withdrawal_state_succeeded.**RevenueWithdrawalStateSucceeded**(**, type: Literal[RevenueWithdra = RevenueWithdrawalStateType.SUCCEEDED, date: _datetime_serializer, return_type=int, when_used=unless-none)], url: str, **extra_data: Any*)

The withdrawal succeeded.

Source: https://core.telegram.org/bots/api#revenuewithdrawalstatesucceeded

type: Literal[RevenueWithdrawalStateType.SUCCEEDED]

Type of the state, always 'succeeded'

date: DateTime

Date the withdrawal was completed in Unix time

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

url: str

An HTTPS URL that can be used to see transaction details

ShippingAddress

class `aiogram.types.shipping_address.`**`ShippingAddress`**(**, country_code: str, state: str, city: str, street_line1: str, street_line2: str, post_code: str, **extra_data: Any*)

This object represents a shipping address.

Source: https://core.telegram.org/bots/api#shippingaddress

`country_code: str`

Two-letter ISO 3166-1 alpha-2 country code

`state: str`

State, if applicable

`city: str`

City

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`street_line1: str`

First line for the address

`street_line2: str`

Second line for the address

`post_code: str`

Address post code

ShippingOption

class `aiogram.types.shipping_option.`**`ShippingOption`**(**, id: str, title: str, prices: List[LabeledPrice], **extra_data: Any*)

This object represents one shipping option.

Source: https://core.telegram.org/bots/api#shippingoption

`id: str`

Shipping option identifier

`title: str`

Option title

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`prices: List[`*`LabeledPrice`*`]`

List of price portions

ShippingQuery

class `aiogram.types.shipping_query.`**`ShippingQuery`**(**, id: str, from_user: User, invoice_payload: str, shipping_address: ShippingAddress, **extra_data: Any*)

This object contains information about an incoming shipping query.

Source: https://core.telegram.org/bots/api#shippingquery

`id: str`

Unique query identifier

`from_user:` *`User`*

User who sent the query

`invoice_payload: str`

Bot-specified invoice payload

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`shipping_address:` *`ShippingAddress`*

User specified shipping address

`answer`(*ok: bool, shipping_options: List[ShippingOption] | None = None, error_message: str | None = None, **kwargs: Any*) → *AnswerShippingQuery*

Shortcut for method *`aiogram.methods.answer_shipping_query.AnswerShippingQuery`* will automatically fill method attributes:

- `shipping_query_id`

If you sent an invoice requesting a shipping address and the parameter *is_flexible* was specified, the Bot API will send an *`aiogram.types.update.Update`* with a *shipping_query* field to the bot. Use this method to reply to shipping queries. On success, `True` is returned.

Source: https://core.telegram.org/bots/api#answershippingquery

Parameters

- **`ok`** – Pass `True` if delivery to the specified address is possible and `False` if there are any problems (for example, if delivery to the specified address is not possible)
- **`shipping_options`** – Required if *ok* is `True`. A JSON-serialized array of available shipping options.
- **`error_message`** – Required if *ok* is `False`. Error message in human readable form that explains why it is impossible to complete the order (e.g. "Sorry, delivery to your desired address is unavailable'). Telegram will display this message to the user.

Returns

instance of method *`aiogram.methods.answer_shipping_query.AnswerShippingQuery`*

StarTransaction

class aiogram.types.star_transaction.**StarTransaction**(**, id: str, amount: int, date: _datetime_serializer, return_type=int, when_used=unless - none)], source:* TransactionPartnerUser | TransactionPartnerFragment | TransactionPartnerTelegramAds | TransactionPartnerOther | *None = None, receiver:* TransactionPartnerUser | TransactionPartnerFragment | TransactionPartnerTelegramAds | TransactionPartnerOther | *None = None, **extra_data: Any*)

Describes a Telegram Star transaction.

Source: https://core.telegram.org/bots/api#startransaction

id: str

Unique identifier of the transaction. Coincides with the identifer of the original transaction for refund transactions. Coincides with *SuccessfulPayment.telegram_payment_charge_id* for successful incoming payments from users.

amount: int

Number of Telegram Stars transferred by the transaction

date: DateTime

Date the transaction was created in Unix time

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

source: *TransactionPartnerUser* **|** *TransactionPartnerFragment* **|** *TransactionPartnerTelegramAds* **|** *TransactionPartnerOther* **| None**

Optional. Source of an incoming transaction (e.g., a user purchasing goods or services, Fragment refunding a failed withdrawal). Only for incoming transactions

receiver: *TransactionPartnerUser* **|** *TransactionPartnerFragment* **|** *TransactionPartnerTelegramAds* **|** *TransactionPartnerOther* **| None**

Optional. Receiver of an outgoing transaction (e.g., a user for a purchase refund, Fragment for a withdrawal). Only for outgoing transactions

StarTransactions

class aiogram.types.star_transactions.**StarTransactions**(**, transactions: List[*StarTransaction*], **extra_data: Any*)

Contains a list of Telegram Star transactions.

Source: https://core.telegram.org/bots/api#startransactions

transactions: List[*StarTransaction***]**

The list of transactions

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

SuccessfulPayment

class aiogram.types.successful_payment.**SuccessfulPayment**(*, *currency: str*, *total_amount: int*, *invoice_payload: str*, *telegram_payment_charge_id: str*, *provider_payment_charge_id: str*, *shipping_option_id: str | None = None*, *order_info: OrderInfo | None = None*, ***extra_data: Any*)

This object contains basic information about a successful payment.

Source: https://core.telegram.org/bots/api#successfulpayment

currency: str

Three-letter ISO 4217 currency code, or 'XTR' for payments in Telegram Stars

total_amount: int

Total price in the *smallest units* of the currency (integer, **not** float/double). For example, for a price of `US$ 1.45` pass `amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies).

invoice_payload: str

Bot-specified invoice payload

telegram_payment_charge_id: str

Telegram payment identifier

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

provider_payment_charge_id: str

Provider payment identifier

shipping_option_id: str | None

Optional. Identifier of the shipping option chosen by the user

order_info: *OrderInfo* | None

Optional. Order information provided by the user

TransactionPartner

class aiogram.types.transaction_partner.**TransactionPartner**(***extra_data: Any*)

This object describes the source of a transaction, or its recipient for outgoing transactions. Currently, it can be one of

- *aiogram.types.transaction_partner_user.TransactionPartnerUser*
- *aiogram.types.transaction_partner_fragment.TransactionPartnerFragment*
- *aiogram.types.transaction_partner_telegram_ads.TransactionPartnerTelegramAds*
- *aiogram.types.transaction_partner_other.TransactionPartnerOther*

Source: https://core.telegram.org/bots/api#transactionpartner

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

TransactionPartnerFragment

class aiogram.types.transaction_partner_fragment.**TransactionPartnerFragment**(**, type: Literal[TransactionPartnerType.FRAG = TransactionPartnerType.FRAGMENT, withdrawal_state:* RevenueWithdrawalStatePending | RevenueWithdrawalStateSucceeded | RevenueWithdrawalStateFailed | *None = None, **extra_data: Any*)

Describes a withdrawal transaction with Fragment.

Source: https://core.telegram.org/bots/api#transactionpartnerfragment

type: Literal[TransactionPartnerType.FRAGMENT]

Type of the transaction partner, always 'fragment'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

withdrawal_state: *RevenueWithdrawalStatePending* **|** *RevenueWithdrawalStateSucceeded* **|** *RevenueWithdrawalStateFailed* **| None**

Optional. State of the transaction if the transaction is outgoing

TransactionPartnerOther

class aiogram.types.transaction_partner_other.**TransactionPartnerOther**(**, type: Literal[TransactionPartnerType.OTHER] = TransactionPartnerType.OTHER, **extra_data: Any*)

Describes a transaction with an unknown source or recipient.

Source: https://core.telegram.org/bots/api#transactionpartnerother

type: Literal[TransactionPartnerType.OTHER]

Type of the transaction partner, always 'other'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

TransactionPartnerTelegramAds

class aiogram.types.transaction_partner_telegram_ads.**TransactionPartnerTelegramAds**(**, type: Literal[TransactionPartnerTy = TransactionPartnerType.TELEGRAM_ADS, **extra_data: Any*)

Describes a withdrawal transaction to the Telegram Ads platform.

Source: https://core.telegram.org/bots/api#transactionpartnertelegramads

type: Literal[TransactionPartnerType.TELEGRAM_ADS]

Type of the transaction partner, always 'telegram_ads'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

TransactionPartnerUser

class aiogram.types.transaction_partner_user.**TransactionPartnerUser**(***, *type: Literal[TransactionPartnerType.USER] = TransactionPartnerType.USER*, *user: User*, *invoice_payload: str | None = None*, *paid_media: List[PaidMediaPreview | PaidMediaPhoto | PaidMediaVideo] | None = None*, *paid_media_payload: str | None = None*, ***extra_data: Any*)

Describes a transaction with a user.

Source: https://core.telegram.org/bots/api#transactionpartneruser

type: Literal[TransactionPartnerType.USER]

Type of the transaction partner, always 'user'

user: *User*

Information about the user

invoice_payload: str | None

Optional. Bot-specified invoice payload

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

paid_media: List[*PaidMediaPreview* | *PaidMediaPhoto* | *PaidMediaVideo*] | None

Optional. Information about the paid media bought by the user

paid_media_payload: str | None

Optional. Bot-specified paid media payload

Telegram Passport

EncryptedCredentials

class aiogram.types.encrypted_credentials.**EncryptedCredentials**(***, *data: str*, *hash: str*, *secret: str*, ***extra_data: Any*)

Describes data required for decrypting and authenticating *aiogram.types.encrypted_passport_element.EncryptedPassportElement*. See the Telegram Passport Documentation for a complete description of the data decryption and authentication processes.

Source: https://core.telegram.org/bots/api#encryptedcredentials

data: str

Base64-encoded encrypted JSON-serialized data with unique user's payload, data hashes and secrets required for *aiogram.types.encrypted_passport_element.EncryptedPassportElement* decryption and authentication

hash: str

Base64-encoded data hash for data authentication

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

secret: str

Base64-encoded secret, encrypted with the bot's public RSA key, required for data decryption

EncryptedPassportElement

class aiogram.types.encrypted_passport_element.**EncryptedPassportElement**(**, type: str, hash: str, data: str | None = None, phone_number: str | None = None, email: str | None = None, files: List[PassportFile] | None = None, front_side: PassportFile | None = None, reverse_side: PassportFile | None = None, selfie: PassportFile | None = None, translation: List[PassportFile] | None = None, **extra_data: Any*)

Describes documents or other Telegram Passport elements shared with the bot by the user.

Source: https://core.telegram.org/bots/api#encryptedpassportelement

type: str

Element type. One of 'personal_details', 'passport', 'driver_license', 'identity_card', 'internal_passport', 'address', 'utility_bill', 'bank_statement', 'rental_agreement', 'passport_registration', 'temporary_registration', 'phone_number', 'email'.

hash: str

Base64-encoded element hash for using in `aiogram.types.passport_element_error_unspecified.PassportElementErrorUnspecified`

data: str | None

Optional. Base64-encoded encrypted Telegram Passport element data provided by the user; available only for 'personal_details', 'passport', 'driver_license', 'identity_card', 'internal_passport' and 'address' types. Can be decrypted and verified using the accompanying `aiogram.types.encrypted_credentials.EncryptedCredentials`.

phone_number: str | None

Optional. User's verified phone number; available only for 'phone_number' type

email: str | None

Optional. User's verified email address; available only for 'email' type

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

files: List[*PassportFile*] | None

Optional. Array of encrypted files with documents provided by the user; available only for 'utility_bill', 'bank_statement', 'rental_agreement', 'passport_registration' and 'temporary_registration' types. Files can be decrypted and verified using the accompanying `aiogram.types.encrypted_credentials.EncryptedCredentials`.

front_side: *PassportFile* | None

Optional. Encrypted file with the front side of the document, provided by the user; available only for 'passport', 'driver_license', 'identity_card' and 'internal_passport'. The file can be decrypted and verified using the accompanying `aiogram.types.encrypted_credentials.EncryptedCredentials`.

reverse_side: *PassportFile* | None

Optional. Encrypted file with the reverse side of the document, provided by the user; available only for 'driver_license' and 'identity_card'. The file can be decrypted and verified using the accompanying `aiogram.types.encrypted_credentials.EncryptedCredentials`.

selfie: *PassportFile* | None

Optional. Encrypted file with the selfie of the user holding a document, provided by the user; available if requested for 'passport', 'driver_license', 'identity_card' and 'internal_passport'. The file can be decrypted and verified using the accompanying `aiogram.types.encrypted_credentials.EncryptedCredentials`.

translation: List[*PassportFile*] | None

Optional. Array of encrypted files with translated versions of documents provided by the user; available if requested for 'passport', 'driver_license', 'identity_card', 'internal_passport', 'utility_bill', 'bank_statement', 'rental_agreement', 'passport_registration' and 'temporary_registration' types. Files can be decrypted and verified using the accompanying `aiogram.types.encrypted_credentials.EncryptedCredentials`.

PassportData

class aiogram.types.passport_data.**PassportData**(**, data: List[EncryptedPassportElement], credentials: EncryptedCredentials, **extra_data: Any*)

Describes Telegram Passport data shared with the bot by the user.

Source: https://core.telegram.org/bots/api#passportdata

data: List[*EncryptedPassportElement*]

Array with information about documents and other Telegram Passport elements that was shared with the bot

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

credentials: *EncryptedCredentials*

Encrypted credentials required to decrypt the data

PassportElementError

class aiogram.types.passport_element_error.**PassportElementError**(***extra_data: Any*)

This object represents an error in the Telegram Passport element which was submitted that should be resolved by the user. It should be one of:

- *aiogram.types.passport_element_error_data_field.PassportElementErrorDataField*
- *aiogram.types.passport_element_error_front_side.PassportElementErrorFrontSide*
- *aiogram.types.passport_element_error_reverse_side.PassportElementErrorReverseSide*
- *aiogram.types.passport_element_error_selfie.PassportElementErrorSelfie*
- *aiogram.types.passport_element_error_file.PassportElementErrorFile*
- *aiogram.types.passport_element_error_files.PassportElementErrorFiles*
- *aiogram.types.passport_element_error_translation_file.PassportElementErrorTranslationFile*
- *aiogram.types.passport_element_error_translation_files.PassportElementErrorTranslationFiles*
- *aiogram.types.passport_element_error_unspecified.PassportElementErrorUnspecified*

Source: https://core.telegram.org/bots/api#passportelementerror

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

PassportElementErrorDataField

class aiogram.types.passport_element_error_data_field.**PassportElementErrorDataField**(*, *source: Literal[PassportElementErr = PassportElementErrorType.DATA*, *type: str*, *field_name: str*, *data_hash: str*, *message: str*, ***extra_data: Any*)

Represents an issue in one of the data fields that was provided by the user. The error is considered resolved when the field's value changes.

Source: https://core.telegram.org/bots/api#passportelementerrordatafield

source: Literal[PassportElementErrorType.DATA]

Error source, must be *data*

type: str

The section of the user's Telegram Passport which has the error, one of 'personal_details', 'passport', 'driver_license', 'identity_card', 'internal_passport', 'address'

field_name: str

Name of the data field which has the error

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

data_hash: str

Base64-encoded data hash

message: str

Error message

PassportElementErrorFile

class aiogram.types.passport_element_error_file.**PassportElementErrorFile**(*, *source: Literal[PassportElementErrorType.FILE] = PassportElementErrorType.FILE*, *type: str*, *file_hash: str*, *message: str*, ***extra_data: Any*)

Represents an issue with a document scan. The error is considered resolved when the file with the document scan changes.

Source: https://core.telegram.org/bots/api#passportelementerrorfile

source: Literal[PassportElementErrorType.FILE]

Error source, must be *file*

type: str

The section of the user's Telegram Passport which has the issue, one of 'utility_bill', 'bank_statement', 'rental_agreement', 'passport_registration', 'temporary_registration'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_hash: str

Base64-encoded file hash

message: str

Error message

PassportElementErrorFiles

class aiogram.types.passport_element_error_files.**PassportElementErrorFiles**(*, *source: Literal[PassportElementErrorType.FILE = PassportElementErrorType.FILES*, *type: str*, *file_hashes: List[str]*, *message: str*, ***extra_data: Any*)

Represents an issue with a list of scans. The error is considered resolved when the list of files containing the scans changes.

Source: https://core.telegram.org/bots/api#passportelementerrorfiles

source: Literal[PassportElementErrorType.FILES]

Error source, must be *files*

type: str

The section of the user's Telegram Passport which has the issue, one of 'utility_bill', 'bank_statement', 'rental_agreement', 'passport_registration', 'temporary_registration'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_hashes: List[str]

List of base64-encoded file hashes

message: str

Error message

PassportElementErrorFrontSide

class aiogram.types.passport_element_error_front_side.**PassportElementErrorFrontSide**(*, *source: Literal[PassportElementErr = PassportElementErrorType.FRONT_SIDE*, *type: str*, *file_hash: str*, *message: str*, ***extra_data: Any*)

Represents an issue with the front side of a document. The error is considered resolved when the file with the front side of the document changes.

Source: https://core.telegram.org/bots/api#passportelementerrorfrontside

source: Literal[PassportElementErrorType.FRONT_SIDE]

Error source, must be *front_side*

type: str

The section of the user's Telegram Passport which has the issue, one of 'passport', 'driver_license', 'identity_card', 'internal_passport'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_hash: str

Base64-encoded hash of the file with the front side of the document

message: str

Error message

PassportElementErrorReverseSide

class aiogram.types.passport_element_error_reverse_side.**PassportElementErrorReverseSide**(**, source: Literal[PassportEleme = PassportElementErrorType.REVERSE_ type: str, file_hash: str, message: str, **extra_data: Any*)

Represents an issue with the reverse side of a document. The error is considered resolved when the file with reverse side of the document changes.

Source: https://core.telegram.org/bots/api#passportelementerrorreverseside

source: Literal[PassportElementErrorType.REVERSE_SIDE]

Error source, must be *reverse_side*

type: str

The section of the user's Telegram Passport which has the issue, one of 'driver_license', 'identity_card'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_hash: str

Base64-encoded hash of the file with the reverse side of the document

message: str

Error message

PassportElementErrorSelfie

class aiogram.types.passport_element_error_selfie.**PassportElementErrorSelfie**(**, source: Literal[PassportElementErrorType.S = PassportElementErrorType.SELFIE, type: str, file_hash: str, message: str, **extra_data: Any*)

Represents an issue with the selfie with a document. The error is considered resolved when the file with the selfie changes.

Source: https://core.telegram.org/bots/api#passportelementerrorselfie

source: Literal[PassportElementErrorType.SELFIE]

Error source, must be *selfie*

type: str

The section of the user's Telegram Passport which has the issue, one of 'passport', 'driver_license', 'identity_card', 'internal_passport'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_hash: str

Base64-encoded hash of the file with the selfie

message: str

Error message

PassportElementErrorTranslationFile

class aiogram.types.passport_element_error_translation_file.**PassportElementErrorTranslationFile**(*, *source: Literal[Pass = PassportElementErrorType.T type: str*, *file_hash: str*, *message: str*, ***extra_data: Any*)

Represents an issue with one of the files that constitute the translation of a document. The error is considered resolved when the file changes.

Source: https://core.telegram.org/bots/api#passportelementerrortranslationfile

source: Literal[PassportElementErrorType.TRANSLATION_FILE]

Error source, must be *translation_file*

type: str

Type of element of the user's Telegram Passport which has the issue, one of 'passport', 'driver_license', 'identity_card', 'internal_passport', 'utility_bill', 'bank_statement', 'rental_agreement', 'passport_registration', 'temporary_registration'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_hash: str

Base64-encoded file hash

message: str

Error message

PassportElementErrorTranslationFiles

class aiogram.types.passport_element_error_translation_files.**PassportElementErrorTranslationFiles**(*, *source: Literal[Pa = PassportElementErrorType type: str*, *file_ha List[str mes-sage: str*, ***extra_dat Any*)

Represents an issue with the translated version of a document. The error is considered resolved when a file with the document translation change.

Source: https://core.telegram.org/bots/api#passportelementerrortranslationfiles

source: Literal[PassportElementErrorType.TRANSLATION_FILES]

Error source, must be *translation_files*

type: str

Type of element of the user's Telegram Passport which has the issue, one of 'passport', 'driver_license', 'identity_card', 'internal_passport', 'utility_bill', 'bank_statement', 'rental_agreement', 'passport_registration', 'temporary_registration'

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

file_hashes: List[str]

List of base64-encoded file hashes

message: str

Error message

PassportElementErrorUnspecified

class aiogram.types.passport_element_error_unspecified.**PassportElementErrorUnspecified**(*, *source: Literal[PassportElemen = PassportElementErrorType.UNSPECIFI type: str, element_hash: str, message: str, **extra_data: Any*)

Represents an issue in an unspecified place. The error is considered resolved when new data is added.

Source: https://core.telegram.org/bots/api#passportelementerrorunspecified

source: Literal[PassportElementErrorType.UNSPECIFIED]

Error source, must be *unspecified*

type: str

Type of element of the user's Telegram Passport which has the issue

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

element_hash: str

Base64-encoded element hash

message: str

Error message

PassportFile

`class aiogram.types.passport_file.PassportFile`(**, file_id: str, file_unique_id: str, file_size: int, file_date: _datetime_serializer, return_type=int, when_used=unless - none)], **extra_data: Any*)

This object represents a file uploaded to Telegram Passport. Currently all Telegram Passport files are in JPEG format when decrypted and don't exceed 10MB.

Source: https://core.telegram.org/bots/api#passportfile

`file_id: str`

Identifier for this file, which can be used to download or reuse the file

`file_unique_id: str`

Unique identifier for this file, which is supposed to be the same over time and for different bots. Can't be used to download or reuse the file.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`file_size: int`

File size in bytes

`file_date: DateTime`

Unix time when the file was uploaded

Getting updates

Update

`class aiogram.types.update.Update`(**, update_id: int, message: Message | None = None, edited_message: Message | None = None, channel_post: Message | None = None, edited_channel_post: Message | None = None, business_connection: BusinessConnection | None = None, business_message: Message | None = None, edited_business_message: Message | None = None, deleted_business_messages: BusinessMessagesDeleted | None = None, message_reaction: MessageReactionUpdated | None = None, message_reaction_count: MessageReactionCountUpdated | None = None, inline_query: InlineQuery | None = None, chosen_inline_result: ChosenInlineResult | None = None, callback_query: CallbackQuery | None = None, shipping_query: ShippingQuery | None = None, pre_checkout_query: PreCheckoutQuery | None = None, purchased_paid_media: PaidMediaPurchased | None = None, poll: Poll | None = None, poll_answer: PollAnswer | None = None, my_chat_member: ChatMemberUpdated | None = None, chat_member: ChatMemberUpdated | None = None, chat_join_request: ChatJoinRequest | None = None, chat_boost: ChatBoostUpdated | None = None, removed_chat_boost: ChatBoostRemoved | None = None, **extra_data: Any*)

This object represents an incoming update.

At most **one** of the optional parameters can be present in any given update.

Source: https://core.telegram.org/bots/api#update

update_id: int

The update's unique identifier. Update identifiers start from a certain positive number and increase sequentially. This identifier becomes especially handy if you're using webhooks, since it allows you to ignore repeated updates or to restore the correct update sequence, should they get out of order. If there are no new updates for at least a week, then identifier of the next update will be chosen randomly instead of sequentially.

message: *Message* **| None**

Optional. New incoming message of any kind - text, photo, sticker, etc.

edited_message: *Message* **| None**

Optional. New version of a message that is known to the bot and was edited. This update may at times be triggered by changes to message fields that are either unavailable or not actively used by your bot.

channel_post: *Message* **| None**

Optional. New incoming channel post of any kind - text, photo, sticker, etc.

edited_channel_post: *Message* **| None**

Optional. New version of a channel post that is known to the bot and was edited. This update may at times be triggered by changes to message fields that are either unavailable or not actively used by your bot.

business_connection: *BusinessConnection* **| None**

Optional. The bot was connected to or disconnected from a business account, or a user edited an existing connection with the bot

business_message: *Message* **| None**

Optional. New message from a connected business account

edited_business_message: *Message* **| None**

Optional. New version of a message from a connected business account

deleted_business_messages: *BusinessMessagesDeleted* **| None**

Optional. Messages were deleted from a connected business account

message_reaction: *MessageReactionUpdated* **| None**

Optional. A reaction to a message was changed by a user. The bot must be an administrator in the chat and must explicitly specify `"message_reaction"` in the list of *allowed_updates* to receive these updates. The update isn't received for reactions set by bots.

message_reaction_count: *MessageReactionCountUpdated* **| None**

Optional. Reactions to a message with anonymous reactions were changed. The bot must be an administrator in the chat and must explicitly specify `"message_reaction_count"` in the list of *allowed_updates* to receive these updates. The updates are grouped and can be sent with delay up to a few minutes.

inline_query: *InlineQuery* **| None**

Optional. New incoming inline query

chosen_inline_result: *ChosenInlineResult* **| None**

Optional. The result of an inline query that was chosen by a user and sent to their chat partner. Please see our documentation on the feedback collecting for details on how to enable these updates for your bot.

callback_query: *CallbackQuery* **| None**

Optional. New incoming callback query

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

shipping_query: *ShippingQuery* **| None**

Optional. New incoming shipping query. Only for invoices with flexible price

pre_checkout_query: *PreCheckoutQuery* **| None**

Optional. New incoming pre-checkout query. Contains full information about checkout

purchased_paid_media: *PaidMediaPurchased* **| None**

Optional. A user purchased paid media with a non-empty payload sent by the bot in a non-channel chat

poll: *Poll* **| None**

Optional. New poll state. Bots receive only updates about manually stopped polls and polls, which are sent by the bot

poll_answer: *PollAnswer* **| None**

Optional. A user changed their answer in a non-anonymous poll. Bots receive new votes only in polls that were sent by the bot itself.

my_chat_member: *ChatMemberUpdated* **| None**

Optional. The bot's chat member status was updated in a chat. For private chats, this update is received only when the bot is blocked or unblocked by the user.

chat_member: *ChatMemberUpdated* **| None**

Optional. A chat member's status was updated in a chat. The bot must be an administrator in the chat and must explicitly specify `"chat_member"` in the list of *allowed_updates* to receive these updates.

chat_join_request: *ChatJoinRequest* **| None**

Optional. A request to join the chat has been sent. The bot must have the *can_invite_users* administrator right in the chat to receive these updates.

chat_boost: *ChatBoostUpdated* **| None**

Optional. A chat boost was added or changed. The bot must be an administrator in the chat to receive these updates.

removed_chat_boost: *ChatBoostRemoved* **| None**

Optional. A boost was removed from a chat. The bot must be an administrator in the chat to receive these updates.

property event_type: str

Detect update type If update type is unknown, raise UpdateTypeLookupError

Returns

property event: TelegramObject

exception `aiogram.types.update.`**UpdateTypeLookupError**

Update does not contain any known event type.

WebhookInfo

class aiogram.types.webhook_info.**WebhookInfo**(**, url: str, has_custom_certificate: bool, pending_update_count: int, ip_address: str | None = None, last_error_date: _datetime_serializer, return_type=int, when_used=unless - none)] | None = None, last_error_message: str | None = None, last_synchronization_error_date: _datetime_serializer, return_type=int, when_used=unless - none)] | None = None, max_connections: int | None = None, allowed_updates: List[str] | None = None, **extra_data: Any*)

Describes the current status of a webhook.

Source: https://core.telegram.org/bots/api#webhookinfo

url: str

Webhook URL, may be empty if webhook is not set up

has_custom_certificate: bool

`True`, if a custom certificate was provided for webhook certificate checks

pending_update_count: int

Number of updates awaiting delivery

ip_address: str | None

Optional. Currently used webhook IP address

last_error_date: DateTime | None

Optional. Unix time for the most recent error that happened when trying to deliver an update via webhook

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

last_error_message: str | None

Optional. Error message in human-readable format for the most recent error that happened when trying to deliver an update via webhook

last_synchronization_error_date: DateTime | None

Optional. Unix time of the most recent error that happened when trying to synchronize available updates with Telegram datacenters

max_connections: int | None

Optional. The maximum allowed number of simultaneous HTTPS connections to the webhook for update delivery

allowed_updates: List[str] | None

Optional. A list of update types the bot is subscribed to. Defaults to all update types except *chat_member*

Games

CallbackGame

class `aiogram.types.callback_game.`**`CallbackGame`**(***extra_data: Any*)

A placeholder, currently holds no information. Use BotFather to set up your game.

Source: https://core.telegram.org/bots/api#callbackgame

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Game

class `aiogram.types.game.`**`Game`**(***, *title: str*, *description: str*, *photo: List[PhotoSize]*, *text: str | None = None*, *text_entities: List[MessageEntity] | None = None*, *animation: Animation | None = None*, ***extra_data: Any*)

This object represents a game. Use BotFather to create and edit games, their short names will act as unique identifiers.

Source: https://core.telegram.org/bots/api#game

`title: str`

Title of the game

`description: str`

Description of the game

`photo: List[PhotoSize]`

Photo that will be displayed in the game message in chats.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`text: str | None`

Optional. Brief description of the game or high scores included in the game message. Can be automatically edited to include current high scores for the game when the bot calls `aiogram.methods.set_game_score.SetGameScore`, or manually edited using `aiogram.methods.edit_message_text.EditMessageText`. 0-4096 characters.

`text_entities: List[MessageEntity] | None`

Optional. Special entities that appear in *text*, such as usernames, URLs, bot commands, etc.

`animation: Animation | None`

Optional. Animation that will be displayed in the game message in chats. Upload via BotFather

GameHighScore

class aiogram.types.game_high_score.**GameHighScore**(***, *position: int*, *user:* User, *score: int*, ***extra_data: Any*)

This object represents one row of the high scores table for a game. And that's about all we've got for now.

If you've got any questions, please check out our https://core.telegram.org/bots/faq **Bot FAQ »**

Source: https://core.telegram.org/bots/api#gamehighscore

position: int

Position in high score table for the game

user: *User*

User

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

score: int

Score

2.3.4 Methods

Here is list of all available API methods:

Stickers

addStickerToSet

Returns: `bool`

class aiogram.methods.add_sticker_to_set.**AddStickerToSet**(***, *user_id: int*, *name: str*, *sticker:* InputSticker, ***extra_data: Any*)

Use this method to add a new sticker to a set created by the bot. Emoji sticker sets can have up to 200 stickers. Other sticker sets can have up to 120 stickers. Returns `True` on success.

Source: https://core.telegram.org/bots/api#addstickertoset

user_id: int

User identifier of sticker set owner

name: str

Sticker set name

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sticker: *InputSticker*

A JSON-serialized object with information about the added sticker. If exactly the same sticker had already been added to the set, then the set isn't changed.

Usage

As bot method

```
result: bool = await bot.add_sticker_to_set(...)
```

Method as object

Imports:

- `from aiogram.methods.add_sticker_to_set import AddStickerToSet`
- alias: `from aiogram.methods import AddStickerToSet`

With specific bot

```
result: bool = await bot(AddStickerToSet(...))
```

As reply into Webhook in handler

```
return AddStickerToSet(...)
```

createNewStickerSet

Returns: `bool`

class `aiogram.methods.create_new_sticker_set.`**`CreateNewStickerSet`**(**, user_id: int, name: str, title: str, stickers: List[InputSticker], sticker_type: str | None = None, needs_repainting: bool | None = None, sticker_format: str | None = None, **extra_data: Any*)

Use this method to create a new sticker set owned by a user. The bot will be able to edit the sticker set thus created. Returns `True` on success.

Source: https://core.telegram.org/bots/api#createnewstickerset

user_id: int

User identifier of created sticker set owner

name: str

Short name of sticker set, to be used in `t.me/addstickers/` URLs (e.g., *animals*). Can contain only English letters, digits and underscores. Must begin with a letter, can't contain consecutive underscores and must end in `"_by_<bot_username>"`. `<bot_username>` is case insensitive. 1-64 characters.

title: str

Sticker set title, 1-64 characters

stickers: List[*InputSticker*]

A JSON-serialized list of 1-50 initial stickers to be added to the sticker set

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sticker_type: str | None

Type of stickers in the set, pass 'regular', 'mask', or 'custom_emoji'. By default, a regular sticker set is created.

needs_repainting: bool | None

Pass `True` if stickers in the sticker set must be repainted to the color of text when used in messages, the accent color if used as emoji status, white on chat photos, or another appropriate color based on context; for custom emoji sticker sets only

sticker_format: str | None

Format of stickers in the set, must be one of 'static', 'animated', 'video'

Deprecated since version API:7.2: https://core.telegram.org/bots/api-changelog#march-31-2024

Usage

As bot method

```
result: bool = await bot.create_new_sticker_set(...)
```

Method as object

Imports:

- `from aiogram.methods.create_new_sticker_set import CreateNewStickerSet`
- alias: `from aiogram.methods import CreateNewStickerSet`

With specific bot

```
result: bool = await bot(CreateNewStickerSet(...))
```

As reply into Webhook in handler

```
return CreateNewStickerSet(...)
```

deleteStickerFromSet

Returns: `bool`

class `aiogram.methods.delete_sticker_from_set.`**`DeleteStickerFromSet`**(*, *sticker: str*, ***extra_data: Any*)

Use this method to delete a sticker from a set created by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletestickerfromset

`sticker: str`

File identifier of the sticker

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.delete_sticker_from_set(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_sticker_from_set import DeleteStickerFromSet`
- alias: `from aiogram.methods import DeleteStickerFromSet`

With specific bot

```
result: bool = await bot(DeleteStickerFromSet(...))
```

As reply into Webhook in handler

```
return DeleteStickerFromSet(...)
```

As shortcut from received object

- *aiogram.types.sticker.Sticker.delete_from_set()*

deleteStickerSet

Returns: `bool`

class `aiogram.methods.delete_sticker_set.`**`DeleteStickerSet`**(**, name: str, **extra_data: Any*)

Use this method to delete a sticker set that was created by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletestickerset

name: str

Sticker set name

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.delete_sticker_set(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_sticker_set import DeleteStickerSet`
- alias: `from aiogram.methods import DeleteStickerSet`

With specific bot

```
result: bool = await bot(DeleteStickerSet(...))
```

As reply into Webhook in handler

```
return DeleteStickerSet(...)
```

getCustomEmojiStickers

Returns: `List[Sticker]`

class `aiogram.methods.get_custom_emoji_stickers.`**`GetCustomEmojiStickers`**(*, *custom_emoji_ids: List[str]*, ***extra_data: Any*)

Use this method to get information about custom emoji stickers by their identifiers. Returns an Array of `aiogram.types.sticker.Sticker` objects.

Source: https://core.telegram.org/bots/api#getcustomemojistickers

custom_emoji_ids: List[str]

A JSON-serialized list of custom emoji identifiers. At most 200 custom emoji identifiers can be specified.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: List[Sticker] = await bot.get_custom_emoji_stickers(...)
```

Method as object

Imports:

- `from aiogram.methods.get_custom_emoji_stickers import GetCustomEmojiStickers`
- alias: `from aiogram.methods import GetCustomEmojiStickers`

With specific bot

```
result: List[Sticker] = await bot(GetCustomEmojiStickers(...))
```

getStickerSet

Returns: `StickerSet`

class `aiogram.methods.get_sticker_set.`**`GetStickerSet`**(**, name: str, **extra_data: Any*)

Use this method to get a sticker set. On success, a *`aiogram.types.sticker_set.StickerSet`* object is returned.

Source: https://core.telegram.org/bots/api#getstickerset

`name: str`

Name of the sticker set

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: StickerSet = await bot.get_sticker_set(...)
```

Method as object

Imports:

- `from aiogram.methods.get_sticker_set import GetStickerSet`
- alias: `from aiogram.methods import GetStickerSet`

With specific bot

```
result: StickerSet = await bot(GetStickerSet(...))
```

replaceStickerInSet

Returns: `bool`

class `aiogram.methods.replace_sticker_in_set.`**`ReplaceStickerInSet`**(*, *user_id: int*, *name: str*, *old_sticker: str*, *sticker: InputSticker*, ***extra_data: Any*)

Use this method to replace an existing sticker in a sticker set with a new one. The method is equivalent to calling *aiogram.methods.delete_sticker_from_set.DeleteStickerFromSet*, then *aiogram.methods.add_sticker_to_set.AddStickerToSet*, then *aiogram.methods.set_sticker_position_in_set.SetStickerPositionInSet*. Returns `True` on success.

Source: https://core.telegram.org/bots/api#replacestickerinset

user_id: int

User identifier of the sticker set owner

name: str

Sticker set name

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

old_sticker: str

File identifier of the replaced sticker

sticker: *InputSticker*

A JSON-serialized object with information about the added sticker. If exactly the same sticker had already been added to the set, then the set remains unchanged.

Usage

As bot method

```
result: bool = await bot.replace_sticker_in_set(...)
```

Method as object

Imports:

- `from aiogram.methods.replace_sticker_in_set import ReplaceStickerInSet`
- alias: `from aiogram.methods import ReplaceStickerInSet`

With specific bot

```
result: bool = await bot(ReplaceStickerInSet(...))
```

As reply into Webhook in handler

```
return ReplaceStickerInSet(...)
```

sendSticker

Returns: `Message`

class `aiogram.methods.send_sticker.`**`SendSticker`**(**, chat_id: int | str, sticker: ~aiogram.types.input_file.InputFile | str, business_connection_id: str | None = None, message_thread_id: int | None = None, emoji: str | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send static .WEBP, animated .TGS, or video .WEBM stickers. On success, the sent *`aiogram.types.message.Message`* is returned.

Source: https://core.telegram.org/bots/api#sendsticker

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`sticker: InputFile | str`

Sticker to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a .WEBP sticker from the Internet, or upload a new .WEBP, .TGS, or .WEBM sticker using multipart/form-data. *More information on Sending Files »*. Video and animated stickers can't be sent via an HTTP URL.

`business_connection_id: str | None`

Unique identifier of the business connection on behalf of which the message will be sent

`message_thread_id: int | None`

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

emoji: str | None
Emoji associated with the sticker; only for just uploaded stickers

disable_notification: bool | None
Sends the message silently. Users will receive a notification with no sound.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

protect_content: bool | Default | None
Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None
Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None
Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None
Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None
Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None
If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_sticker(...)
```

Method as object

Imports:

- `from aiogram.methods.send_sticker import SendSticker`
- alias: `from aiogram.methods import SendSticker`

With specific bot

```
result: Message = await bot(SendSticker(...))
```

As reply into Webhook in handler

```
return SendSticker(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_sticker()*
- *aiogram.types.message.Message.reply_sticker()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_sticker()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_sticker_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_sticker()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_sticker()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_sticker()*

setCustomEmojiStickerSetThumbnail

Returns: `bool`

class `aiogram.methods.set_custom_emoji_sticker_set_thumbnail.`**`SetCustomEmojiStickerSetThumbnail`**(**, name: str, custom_emoji_ str | None = None, **extra_data: Any*)

Use this method to set the thumbnail of a custom emoji sticker set. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setcustomemojistickersetthumbnail

`name: str`

Sticker set name

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

custom_emoji_id: str | None

Custom emoji identifier of a sticker from the sticker set; pass an empty string to drop the thumbnail and use the first sticker as the thumbnail.

Usage

As bot method

```
result: bool = await bot.set_custom_emoji_sticker_set_thumbnail(...)
```

Method as object

Imports:

- `from aiogram.methods.set_custom_emoji_sticker_set_thumbnail import SetCustomEmojiStickerSetThumbnail`
- alias: `from aiogram.methods import SetCustomEmojiStickerSetThumbnail`

With specific bot

```
result: bool = await bot(SetCustomEmojiStickerSetThumbnail(...))
```

As reply into Webhook in handler

```
return SetCustomEmojiStickerSetThumbnail(...)
```

setStickerEmojiList

Returns: `bool`

class `aiogram.methods.set_sticker_emoji_list.`**SetStickerEmojiList**(**, sticker: str, emoji_list: List[str], **extra_data: Any*)

Use this method to change the list of emoji assigned to a regular or custom emoji sticker. The sticker must belong to a sticker set created by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setstickeremojilist

sticker: str

File identifier of the sticker

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

emoji_list: List[str]

A JSON-serialized list of 1-20 emoji associated with the sticker

Usage

As bot method

```
result: bool = await bot.set_sticker_emoji_list(...)
```

Method as object

Imports:

- `from aiogram.methods.set_sticker_emoji_list import SetStickerEmojiList`
- alias: `from aiogram.methods import SetStickerEmojiList`

With specific bot

```
result: bool = await bot(SetStickerEmojiList(...))
```

As reply into Webhook in handler

```
return SetStickerEmojiList(...)
```

setStickerKeywords

Returns: `bool`

class `aiogram.methods.set_sticker_keywords.`**`SetStickerKeywords`**(*, *sticker: str*, *keywords: List[str] | None = None*, ***extra_data: Any*)

Use this method to change search keywords assigned to a regular or custom emoji sticker. The sticker must belong to a sticker set created by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setstickerkeywords

`sticker: str`

File identifier of the sticker

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`keywords: List[str] | None`

A JSON-serialized list of 0-20 search keywords for the sticker with total length of up to 64 characters

Usage

As bot method

```
result: bool = await bot.set_sticker_keywords(...)
```

Method as object

Imports:

- `from aiogram.methods.set_sticker_keywords import SetStickerKeywords`
- alias: `from aiogram.methods import SetStickerKeywords`

With specific bot

```
result: bool = await bot(SetStickerKeywords(...))
```

As reply into Webhook in handler

```
return SetStickerKeywords(...)
```

setStickerMaskPosition

Returns: `bool`

class `aiogram.methods.set_sticker_mask_position.`**`SetStickerMaskPosition`**(*, *sticker: str*, *mask_position:* MaskPosition | *None* = *None*, ***extra_data: Any*)

Use this method to change the mask position of a mask sticker. The sticker must belong to a sticker set that was created by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setstickermaskposition

`sticker: str`

File identifier of the sticker

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`mask_position:` ***`MaskPosition`*** **`| None`**

A JSON-serialized object with the position where the mask should be placed on faces. Omit the parameter to remove the mask position.

Usage

As bot method

```
result: bool = await bot.set_sticker_mask_position(...)
```

Method as object

Imports:

- `from aiogram.methods.set_sticker_mask_position import SetStickerMaskPosition`
- alias: `from aiogram.methods import SetStickerMaskPosition`

With specific bot

```
result: bool = await bot(SetStickerMaskPosition(...))
```

As reply into Webhook in handler

```
return SetStickerMaskPosition(...)
```

setStickerPositionInSet

Returns: `bool`

class `aiogram.methods.set_sticker_position_in_set.`**`SetStickerPositionInSet`**(*, *sticker: str*, *position: int*, ***extra_data: Any*)

Use this method to move a sticker in a set created by the bot to a specific position. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setstickerpositioninset

sticker: str

File identifier of the sticker

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

position: int

New sticker position in the set, zero-based

Usage

As bot method

```
result: bool = await bot.set_sticker_position_in_set(...)
```

Method as object

Imports:

- `from aiogram.methods.set_sticker_position_in_set import SetStickerPositionInSet`
- alias: `from aiogram.methods import SetStickerPositionInSet`

With specific bot

```
result: bool = await bot(SetStickerPositionInSet(...))
```

As reply into Webhook in handler

```
return SetStickerPositionInSet(...)
```

As shortcut from received object

- *aiogram.types.sticker.Sticker.set_position_in_set()*

setStickerSetThumbnail

Returns: `bool`

class `aiogram.methods.set_sticker_set_thumbnail.`**`SetStickerSetThumbnail`**(**, name: str, user_id: int, format: str, thumbnail: InputFile | str | None = None, **extra_data: Any*)

Use this method to set the thumbnail of a regular or mask sticker set. The format of the thumbnail file must match the format of the stickers in the set. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setstickersetthumbnail

name: str

Sticker set name

user_id: int

User identifier of the sticker set owner

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

format: str

Format of the thumbnail, must be one of 'static' for a **.WEBP** or **.PNG** image, 'animated' for a **.TGS** animation, or 'video' for a **WEBM** video

thumbnail: *InputFile* **| str | None**

A **.WEBP** or **.PNG** image with the thumbnail, must be up to 128 kilobytes in size and have a width and height of exactly 100px, or a **.TGS** animation with a thumbnail up to 32 kilobytes in size (see https://core.telegram.org/stickers#animation-requirements <https://core.telegram.org/stickers#animation-requirements>`_`https://core.telegram.org/stickers#animation-requirements for animated sticker technical requirements), or a **WEBM** video with the thumbnail up to 32 kilobytes in size; see https://core.telegram.org/stickers#video-requirements <https://core.telegram.org/stickers#video-requirements>`_`https://core.telegram.org/stickers#video-requirements for video sticker technical requirements. Pass a *file_id* as a String to send a file that already exists on the Telegram servers, pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*. Animated and video sticker set thumbnails can't be uploaded via HTTP URL. If omitted, then the thumbnail is dropped and the first sticker is used as the thumbnail.

Usage

As bot method

```
result: bool = await bot.set_sticker_set_thumbnail(...)
```

Method as object

Imports:

- `from aiogram.methods.set_sticker_set_thumbnail import SetStickerSetThumbnail`
- alias: `from aiogram.methods import SetStickerSetThumbnail`

With specific bot

```
result: bool = await bot(SetStickerSetThumbnail(...))
```

As reply into Webhook in handler

```
return SetStickerSetThumbnail(...)
```

setStickerSetTitle

Returns: `bool`

class `aiogram.methods.set_sticker_set_title.`**SetStickerSetTitle**(**, name: str, title: str, **extra_data: Any*)

Use this method to set the title of a created sticker set. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setstickersettitle

name: str

Sticker set name

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

title: str

Sticker set title, 1-64 characters

Usage

As bot method

```
result: bool = await bot.set_sticker_set_title(...)
```

Method as object

Imports:

- `from aiogram.methods.set_sticker_set_title import SetStickerSetTitle`
- alias: `from aiogram.methods import SetStickerSetTitle`

With specific bot

```
result: bool = await bot(SetStickerSetTitle(...))
```

As reply into Webhook in handler

```
return SetStickerSetTitle(...)
```

uploadStickerFile

Returns: `File`

class `aiogram.methods.upload_sticker_file.`**`UploadStickerFile`**(**, user_id: int, sticker:* InputFile, *sticker_format: str, **extra_data: Any*)

Use this method to upload a file with a sticker for later use in the *aiogram.methods.create_new_sticker_set.CreateNewStickerSet*, *aiogram.methods.add_sticker_to_set.AddStickerToSet*, or *aiogram.methods.replace_sticker_in_set.ReplaceStickerInSet* methods (the file can be used multiple times). Returns the uploaded *aiogram.types.file.File* on success.

Source: https://core.telegram.org/bots/api#uploadstickerfile

user_id: int

User identifier of sticker file owner

sticker: ***InputFile***

A file with the sticker in .WEBP, .PNG, .TGS, or .WEBM format. See https://core.telegram.org/stickers <https://core.telegram.org/stickers>`_`https://core.telegram.org/stickers for technical requirements. *More information on Sending Files »*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sticker_format: str

Format of the sticker, must be one of 'static', 'animated', 'video'

Usage

As bot method

```
result: File = await bot.upload_sticker_file(...)
```

Method as object

Imports:

- `from aiogram.methods.upload_sticker_file import UploadStickerFile`
- alias: `from aiogram.methods import UploadStickerFile`

With specific bot

```
result: File = await bot(UploadStickerFile(...))
```

Available methods

answerCallbackQuery

Returns: bool

class aiogram.methods.answer_callback_query.**AnswerCallbackQuery**(**, callback_query_id: str, text: str | None = None, show_alert: bool | None = None, url: str | None = None, cache_time: int | None = None, **extra_data: Any*)

Use this method to send answers to callback queries sent from inline keyboards. The answer will be displayed to the user as a notification at the top of the chat screen or as an alert. On success, `True` is returned.

> Alternatively, the user can be redirected to the specified Game URL. For this option to work, you must first create a game for your bot via @BotFather and accept the terms. Otherwise, you may use links like `t.me/your_bot?start=XXXX` that open your bot with a parameter.

Source: https://core.telegram.org/bots/api#answercallbackquery

callback_query_id: str

Unique identifier for the query to be answered

text: str | None

Text of the notification. If not specified, nothing will be shown to the user, 0-200 characters

show_alert: bool | None

If `True`, an alert will be shown by the client instead of a notification at the top of the chat screen. Defaults to *false*.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

url: str | None

URL that will be opened by the user's client. If you have created a *aiogram.types.game.Game* and accepted the conditions via @BotFather, specify the URL that opens your game - note that this will only work if the query comes from a https://core.telegram.org/bots/api#inlinekeyboardbutton *callback_game* button.

cache_time: int | None

The maximum amount of time in seconds that the result of the callback query may be cached client-side. Telegram apps will support caching starting in version 3.14. Defaults to 0.

Usage

As bot method

```
result: bool = await bot.answer_callback_query(...)
```

Method as object

Imports:

- `from aiogram.methods.answer_callback_query import AnswerCallbackQuery`
- alias: `from aiogram.methods import AnswerCallbackQuery`

With specific bot

```
result: bool = await bot(AnswerCallbackQuery(...))
```

As reply into Webhook in handler

```
return AnswerCallbackQuery(...)
```

As shortcut from received object

- *aiogram.types.callback_query.CallbackQuery.answer()*

approveChatJoinRequest

Returns: `bool`

class `aiogram.methods.approve_chat_join_request.`**`ApproveChatJoinRequest`**(**, chat_id: int | str, user_id: int, **extra_data: Any*)

Use this method to approve a chat join request. The bot must be an administrator in the chat for this to work and must have the *can_invite_users* administrator right. Returns `True` on success.

Source: https://core.telegram.org/bots/api#approvechatjoinrequest

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_id: int

Unique identifier of the target user

Usage

As bot method

```
result: bool = await bot.approve_chat_join_request(...)
```

Method as object

Imports:

- `from aiogram.methods.approve_chat_join_request import ApproveChatJoinRequest`
- alias: `from aiogram.methods import ApproveChatJoinRequest`

With specific bot

```
result: bool = await bot(ApproveChatJoinRequest(...))
```

As reply into Webhook in handler

```
return ApproveChatJoinRequest(...)
```

As shortcut from received object

- *aiogram.types.chat_join_request.ChatJoinRequest.approve()*

banChatMember

Returns: `bool`

class `aiogram.methods.ban_chat_member.`**BanChatMember**(**, chat_id: int | str, user_id: int, until_date: datetime | timedelta | int | None = None, revoke_messages: bool | None = None, **extra_data: Any*)

Use this method to ban a user in a group, a supergroup or a channel. In the case of supergroups and channels, the user will not be able to return to the chat on their own using invite links, etc., unless unbanned first. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#banchatmember

chat_id: int | str

Unique identifier for the target group or username of the target supergroup or channel (in the format `@channelusername`)

user_id: int

Unique identifier of the target user

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

until_date: datetime.datetime | datetime.timedelta | int | None

Date when the user will be unbanned; Unix time. If user is banned for more than 366 days or less than 30 seconds from the current time they are considered to be banned forever. Applied for supergroups and channels only.

revoke_messages: bool | None

Pass `True` to delete all messages from the chat for the user that is being removed. If `False`, the user will be able to see messages in the group that were sent before the user was removed. Always `True` for supergroups and channels.

Usage

As bot method

```
result: bool = await bot.ban_chat_member(...)
```

Method as object

Imports:

- `from aiogram.methods.ban_chat_member import BanChatMember`
- alias: `from aiogram.methods import BanChatMember`

With specific bot

```
result: bool = await bot(BanChatMember(...))
```

As reply into Webhook in handler

```
return BanChatMember(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.ban()*

banChatSenderChat

Returns: `bool`

class `aiogram.methods.ban_chat_sender_chat.`**BanChatSenderChat**(*, *chat_id: int | str*, *sender_chat_id: int*, ***extra_data: Any*)

Use this method to ban a channel chat in a supergroup or a channel. Until the chat is unbanned, the owner of the banned chat won't be able to send messages on behalf of **any of their channels**. The bot must be an administrator in the supergroup or channel for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#banchatsenderchat

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sender_chat_id: int

Unique identifier of the target sender chat

Usage

As bot method

```
result: bool = await bot.ban_chat_sender_chat(...)
```

Method as object

Imports:

- `from aiogram.methods.ban_chat_sender_chat import BanChatSenderChat`
- alias: `from aiogram.methods import BanChatSenderChat`

With specific bot

```
result: bool = await bot(BanChatSenderChat(...))
```

As reply into Webhook in handler

```
return BanChatSenderChat(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.ban_sender_chat()*

close

Returns: bool

class aiogram.methods.close.**Close**(***extra_data: Any*)

Use this method to close the bot instance before moving it from one local server to another. You need to delete the webhook before calling this method to ensure that the bot isn't launched again after server restart. The method will return error 429 in the first 10 minutes after the bot is launched. Returns True on success. Requires no parameters.

Source: https://core.telegram.org/bots/api#close

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.close(...)
```

Method as object

Imports:

- from aiogram.methods.close import Close
- alias: from aiogram.methods import Close

With specific bot

```
result: bool = await bot(Close(...))
```

As reply into Webhook in handler

```
return Close(...)
```

closeForumTopic

Returns: bool

class aiogram.methods.close_forum_topic.**CloseForumTopic**(*, *chat_id: int | str*, *message_thread_id: int*, ***extra_data: Any*)

Use this method to close an open topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights, unless it is the creator of the topic. Returns True on success.

Source: https://core.telegram.org/bots/api#closeforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_thread_id: int

Unique identifier for the target message thread of the forum topic

Usage

As bot method

```
result: bool = await bot.close_forum_topic(...)
```

Method as object

Imports:

- from aiogram.methods.close_forum_topic import CloseForumTopic
- alias: from aiogram.methods import CloseForumTopic

With specific bot

```
result: bool = await bot(CloseForumTopic(...))
```

As reply into Webhook in handler

```
return CloseForumTopic(...)
```

closeGeneralForumTopic

Returns: `bool`

class `aiogram.methods.close_general_forum_topic.`**CloseGeneralForumTopic**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method to close an open 'General' topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#closegeneralforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.close_general_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.close_general_forum_topic import CloseGeneralForumTopic`
- alias: `from aiogram.methods import CloseGeneralForumTopic`

With specific bot

```
result: bool = await bot(CloseGeneralForumTopic(...))
```

As reply into Webhook in handler

```
return CloseGeneralForumTopic(...)
```

copyMessage

Returns: `MessageId`

class `aiogram.methods.copy_message.`**`CopyMessage`**(**, chat_id: int | str, from_chat_id: int | str, message_id: int, message_thread_id: int | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to copy messages of any kind. Service messages, paid media messages, giveaway messages, giveaway winners messages, and invoice messages can't be copied. A quiz `aiogram.methods.poll.Poll` can be copied only if the value of the field *correct_option_id* is known to the bot. The method is analogous to the method *`aiogram.methods.forward_message.ForwardMessage`*, but the copied message doesn't have a link to the original message. Returns the *`aiogram.types.message_id.MessageId`* of the sent message on success.

Source: https://core.telegram.org/bots/api#copymessage

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`from_chat_id: int | str`

Unique identifier for the chat where the original message was sent (or channel username in the format `@channelusername`)

message_id: int

Message identifier in the chat specified in *from_chat_id*

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

caption: str | None

New caption for media, 0-1024 characters after entities parsing. If not specified, the original caption is kept

parse_mode: str | Default | None

Mode for parsing entities in the new caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the new caption, which can be specified instead of *parse_mode*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

show_caption_above_media: bool | Default | None

Pass `True`, if the caption must be shown above the message media. Ignored if a new caption isn't specified.

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: MessageId = await bot.copy_message(...)
```

Method as object

Imports:

- `from aiogram.methods.copy_message import CopyMessage`
- alias: `from aiogram.methods import CopyMessage`

With specific bot

```
result: MessageId = await bot(CopyMessage(...))
```

As reply into Webhook in handler

```
return CopyMessage(...)
```

As shortcut from received object

- *`aiogram.types.message.Message.copy_to()`*

copyMessages

Returns: `List[MessageId]`

class `aiogram.methods.copy_messages.`**`CopyMessages`**(**, chat_id: int | str, from_chat_id: int | str, message_ids: List[int], message_thread_id: int | None = None, disable_notification: bool | None = None, protect_content: bool | None = None, remove_caption: bool | None = None, **extra_data: Any*)

Use this method to copy messages of any kind. If some of the specified messages can't be found or copied, they are skipped. Service messages, paid media messages, giveaway messages, giveaway winners messages, and invoice messages can't be copied. A quiz `aiogram.methods.poll.Poll` can be copied only if the value of the field *correct_option_id* is known to the bot. The method is analogous to the method *`aiogram.methods.forward_messages.ForwardMessages`*, but the copied messages don't have a link to the original message. Album grouping is kept for copied messages. On success, an array of *`aiogram.types.message_id.MessageId`* of the sent messages is returned.

Source: https://core.telegram.org/bots/api#copymessages

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

from_chat_id: int | str

Unique identifier for the chat where the original messages were sent (or channel username in the format `@channelusername`)

message_ids: List[int]

A JSON-serialized list of 1-100 identifiers of messages in the chat *from_chat_id* to copy. The identifiers must be specified in a strictly increasing order.

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

disable_notification: bool | None

Sends the messages silently. Users will receive a notification with no sound.

protect_content: bool | None

Protects the contents of the sent messages from forwarding and saving

remove_caption: bool | None

Pass `True` to copy the messages without their captions

Usage

As bot method

```
result: List[MessageId] = await bot.copy_messages(...)
```

Method as object

Imports:

- `from aiogram.methods.copy_messages import CopyMessages`
- alias: `from aiogram.methods import CopyMessages`

With specific bot

```
result: List[MessageId] = await bot(CopyMessages(...))
```

As reply into Webhook in handler

```
return CopyMessages(...)
```

createChatInviteLink

Returns: `ChatInviteLink`

class `aiogram.methods.create_chat_invite_link.`**`CreateChatInviteLink`**(**, chat_id: int | str, name: str | None = None, expire_date: datetime | timedelta | int | None = None, member_limit: int | None = None, creates_join_request: bool | None = None, **extra_data: Any*)

Use this method to create an additional invite link for a chat. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. The link can be revoked using the method *`aiogram.methods.revoke_chat_invite_link.RevokeChatInviteLink`*. Returns the new invite link as *`aiogram.types.chat_invite_link.ChatInviteLink`* object.

Source: https://core.telegram.org/bots/api#createchatinvitelink

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`name: str | None`

Invite link name; 0-32 characters

`expire_date: datetime.datetime | datetime.timedelta | int | None`

Point in time (Unix timestamp) when the link will expire

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`member_limit: int | None`

The maximum number of users that can be members of the chat simultaneously after joining the chat via this invite link; 1-99999

`creates_join_request: bool | None`

`True`, if users joining the chat via the link need to be approved by chat administrators. If `True`, *member_limit* can't be specified

Usage

As bot method

```
result: ChatInviteLink = await bot.create_chat_invite_link(...)
```

Method as object

Imports:

- `from aiogram.methods.create_chat_invite_link import CreateChatInviteLink`
- alias: `from aiogram.methods import CreateChatInviteLink`

With specific bot

```
result: ChatInviteLink = await bot(CreateChatInviteLink(...))
```

As reply into Webhook in handler

```
return CreateChatInviteLink(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.create_invite_link()*

createChatSubscriptionInviteLink

Returns: `ChatInviteLink`

class `aiogram.methods.create_chat_subscription_invite_link.`**`CreateChatSubscriptionInviteLink`**(**, chat_id: int | str, subscription_period: datetime | timedelta | int, subscription_price: int, name: str | None = None, **extra_data: Any*)

Use this method to create a subscription invite link for a channel chat. The bot must have the *can_invite_users* administrator rights. The link can be edited using the method *`aiogram.methods.edit_chat_subscription_invite_link.EditChatSubscriptionInviteLink`* or revoked using the method *`aiogram.methods.revoke_chat_invite_link.RevokeChatInviteLink`*. Returns the new invite link as a *`aiogram.types.chat_invite_link.ChatInviteLink`* object.

Source: https://core.telegram.org/bots/api#createchatsubscriptioninvitelink

`chat_id: int | str`

Unique identifier for the target channel chat or username of the target channel (in the format `@channelusername`)

`subscription_period: datetime.datetime | datetime.timedelta | int`

The number of seconds the subscription will be active for before the next payment. Currently, it must always be 2592000 (30 days).

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`subscription_price: int`

The amount of Telegram Stars a user must pay initially and after each subsequent subscription period to be a member of the chat; 1-2500

name: str | None

Invite link name; 0-32 characters

Usage

As bot method

```
result: ChatInviteLink = await bot.create_chat_subscription_invite_link(...)
```

Method as object

Imports:

- `from aiogram.methods.create_chat_subscription_invite_link import CreateChatSubscriptionInviteLink`
- alias: `from aiogram.methods import CreateChatSubscriptionInviteLink`

With specific bot

```
result: ChatInviteLink = await bot(CreateChatSubscriptionInviteLink(...))
```

As reply into Webhook in handler

```
return CreateChatSubscriptionInviteLink(...)
```

createForumTopic

Returns: `ForumTopic`

class `aiogram.methods.create_forum_topic.`**CreateForumTopic**(**, chat_id: int | str, name: str, icon_color: int | None = None, icon_custom_emoji_id: str | None = None, **extra_data: Any*)

Use this method to create a topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights. Returns information about the created topic as a `aiogram.types.forum_topic.ForumTopic` object.

Source: https://core.telegram.org/bots/api#createforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

name: str

Topic name, 1-128 characters

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

icon_color: int | None

Color of the topic icon in RGB format. Currently, must be one of 7322096 (0x6FB9F0), 16766590 (0xFFD67E), 13338331 (0xCB86DB), 9367192 (0x8EEE98), 16749490 (0xFF93B2), or 16478047 (0xFB6F5F)

icon_custom_emoji_id: str | None

Unique identifier of the custom emoji shown as the topic icon. Use `aiogram.methods.get_forum_topic_icon_stickers.GetForumTopicIconStickers` to get all allowed custom emoji identifiers.

Usage

As bot method

```
result: ForumTopic = await bot.create_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.create_forum_topic import CreateForumTopic`
- alias: `from aiogram.methods import CreateForumTopic`

With specific bot

```
result: ForumTopic = await bot(CreateForumTopic(...))
```

As reply into Webhook in handler

```
return CreateForumTopic(...)
```

declineChatJoinRequest

Returns: `bool`

class `aiogram.methods.decline_chat_join_request.`**DeclineChatJoinRequest**(*, *chat_id: int | str*, *user_id: int*, ***extra_data: Any*)

Use this method to decline a chat join request. The bot must be an administrator in the chat for this to work and must have the *can_invite_users* administrator right. Returns `True` on success.

Source: https://core.telegram.org/bots/api#declinechatjoinrequest

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_id: int

Unique identifier of the target user

Usage

As bot method

```
result: bool = await bot.decline_chat_join_request(...)
```

Method as object

Imports:

- `from aiogram.methods.decline_chat_join_request import DeclineChatJoinRequest`
- alias: `from aiogram.methods import DeclineChatJoinRequest`

With specific bot

```
result: bool = await bot(DeclineChatJoinRequest(...))
```

As reply into Webhook in handler

```
return DeclineChatJoinRequest(...)
```

As shortcut from received object

- *aiogram.types.chat_join_request.ChatJoinRequest.decline()*

deleteChatPhoto

Returns: `bool`

class `aiogram.methods.delete_chat_photo.`**`DeleteChatPhoto`**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method to delete a chat photo. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletechatphoto

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.delete_chat_photo(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_chat_photo import DeleteChatPhoto`
- alias: `from aiogram.methods import DeleteChatPhoto`

With specific bot

```
result: bool = await bot(DeleteChatPhoto(...))
```

As reply into Webhook in handler

```
return DeleteChatPhoto(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.delete_photo()*

deleteChatStickerSet

Returns: `bool`

class `aiogram.methods.delete_chat_sticker_set.`**DeleteChatStickerSet**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method to delete a group sticker set from a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Use the field *can_set_sticker_set* optionally returned in *aiogram.methods.get_chat.GetChat* requests to check if the bot can use this method. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletechatstickerset

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.delete_chat_sticker_set(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_chat_sticker_set import DeleteChatStickerSet`
- alias: `from aiogram.methods import DeleteChatStickerSet`

With specific bot

```
result: bool = await bot(DeleteChatStickerSet(...))
```

As reply into Webhook in handler

```
return DeleteChatStickerSet(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.delete_sticker_set()*

deleteForumTopic

Returns: `bool`

class `aiogram.methods.delete_forum_topic.`**`DeleteForumTopic`**(*, *chat_id: int* | *str*, *message_thread_id: int*, ***extra_data: Any*)

Use this method to delete a forum topic along with all its messages in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_delete_messages* administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deleteforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_thread_id: int

Unique identifier for the target message thread of the forum topic

Usage

As bot method

```
result: bool = await bot.delete_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_forum_topic import DeleteForumTopic`
- alias: `from aiogram.methods import DeleteForumTopic`

With specific bot

```
result: bool = await bot(DeleteForumTopic(...))
```

As reply into Webhook in handler

```
return DeleteForumTopic(...)
```

deleteMyCommands

Returns: `bool`

class `aiogram.methods.delete_my_commands.`**`DeleteMyCommands`**(**, scope: BotCommandScopeDefault | BotCommandScopeAllPrivateChats | BotCommandScopeAllGroupChats | BotCommandScopeAllChatAdministrators | BotCommandScopeChat | BotCommandScopeChatAdministrators | BotCommandScopeChatMember | None = None, language_code: str | None = None, **extra_data: Any*)

Use this method to delete the list of the bot's commands for the given scope and user language. After deletion, higher level commands will be shown to affected users. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletemycommands

scope: *BotCommandScopeDefault* | *BotCommandScopeAllPrivateChats* | *BotCommandScopeAllGroupChats* | *BotCommandScopeAllChatAdministrators* | *BotCommandScopeChat* | *BotCommandScopeChatAdministrators* | *BotCommandScopeChatMember* | None

A JSON-serialized object, describing scope of users for which the commands are relevant. Defaults to *`aiogram.types.bot_command_scope_default.BotCommandScopeDefault`*.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

language_code: str | None

A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

Usage

As bot method

```
result: bool = await bot.delete_my_commands(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_my_commands import DeleteMyCommands`
- alias: `from aiogram.methods import DeleteMyCommands`

With specific bot

```
result: bool = await bot(DeleteMyCommands(...))
```

As reply into Webhook in handler

```
return DeleteMyCommands(...)
```

editChatInviteLink

Returns: `ChatInviteLink`

class `aiogram.methods.edit_chat_invite_link.`**`EditChatInviteLink`**(*, *chat_id: int | str*, *invite_link: str*, *name: str | None = None*, *expire_date: datetime | timedelta | int | None = None*, *member_limit: int | None = None*, *creates_join_request: bool | None = None*, ***extra_data: Any*)

Use this method to edit a non-primary invite link created by the bot. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns the edited invite link as a *`aiogram.types.chat_invite_link.ChatInviteLink`* object.

Source: https://core.telegram.org/bots/api#editchatinvitelink

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`invite_link: str`

The invite link to edit

`name: str | None`

Invite link name; 0-32 characters

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

expire_date: datetime.datetime | datetime.timedelta | int | None

Point in time (Unix timestamp) when the link will expire

member_limit: int | None

The maximum number of users that can be members of the chat simultaneously after joining the chat via this invite link; 1-99999

creates_join_request: bool | None

`True`, if users joining the chat via the link need to be approved by chat administrators. If `True`, *member_limit* can't be specified

Usage

As bot method

```
result: ChatInviteLink = await bot.edit_chat_invite_link(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_chat_invite_link import EditChatInviteLink`
- alias: `from aiogram.methods import EditChatInviteLink`

With specific bot

```
result: ChatInviteLink = await bot(EditChatInviteLink(...))
```

As reply into Webhook in handler

```
return EditChatInviteLink(...)
```

As shortcut from received object

- *`aiogram.types.chat.Chat.edit_invite_link()`*

editChatSubscriptionInviteLink

Returns: `ChatInviteLink`

class `aiogram.methods.edit_chat_subscription_invite_link.`**`EditChatSubscriptionInviteLink`**(*, *chat_id: int | str*, *invite_link: str*, *name: str | None = None*, ***extra_data: Any*)

Use this method to edit a subscription invite link created by the bot. The bot must have the *can_invite_users* administrator rights. Returns the edited invite link as a *`aiogram.types.chat_invite_link.ChatInviteLink`* object.

Source: https://core.telegram.org/bots/api#editchatsubscriptioninvitelink

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`invite_link: str`

The invite link to edit

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`name: str | None`

Invite link name; 0-32 characters

Usage

As bot method

```
result: ChatInviteLink = await bot.edit_chat_subscription_invite_link(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_chat_subscription_invite_link import EditChatSubscriptionInviteLink`
- alias: `from aiogram.methods import EditChatSubscriptionInviteLink`

With specific bot

```
result: ChatInviteLink = await bot(EditChatSubscriptionInviteLink(...))
```

As reply into Webhook in handler

```
return EditChatSubscriptionInviteLink(...)
```

editForumTopic

Returns: `bool`

class `aiogram.methods.edit_forum_topic.`**`EditForumTopic`**(*, *chat_id: int | str*, *message_thread_id: int*, *name: str | None = None*, *icon_custom_emoji_id: str | None = None*, ***extra_data: Any*)

Use this method to edit name and icon of a topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights, unless it is the creator of the topic. Returns `True` on success.

Source: https://core.telegram.org/bots/api#editforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

message_thread_id: int

Unique identifier for the target message thread of the forum topic

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

name: str | None

New topic name, 0-128 characters. If not specified or empty, the current name of the topic will be kept

icon_custom_emoji_id: str | None

New unique identifier of the custom emoji shown as the topic icon. Use *`aiogram.methods.get_forum_topic_icon_stickers.GetForumTopicIconStickers`* to get all allowed custom emoji identifiers. Pass an empty string to remove the icon. If not specified, the current icon will be kept

Usage

As bot method

```
result: bool = await bot.edit_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_forum_topic import EditForumTopic`
- alias: `from aiogram.methods import EditForumTopic`

With specific bot

```
result: bool = await bot(EditForumTopic(...))
```

As reply into Webhook in handler

```
return EditForumTopic(...)
```

editGeneralForumTopic

Returns: `bool`

class `aiogram.methods.edit_general_forum_topic.`**`EditGeneralForumTopic`**(**, chat_id: int | str, name: str, **extra_data: Any*)

Use this method to edit the name of the 'General' topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#editgeneralforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

name: str

New topic name, 1-128 characters

Usage

As bot method

```
result: bool = await bot.edit_general_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_general_forum_topic import EditGeneralForumTopic`
- alias: `from aiogram.methods import EditGeneralForumTopic`

With specific bot

```
result: bool = await bot(EditGeneralForumTopic(...))
```

As reply into Webhook in handler

```
return EditGeneralForumTopic(...)
```

exportChatInviteLink

Returns: `str`

class `aiogram.methods.export_chat_invite_link.`**`ExportChatInviteLink`**(**, chat_id: int | str, **extra_data: Any*)

Use this method to generate a new primary invite link for a chat; any previously generated primary link is revoked. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns the new invite link as *String* on success.

Note: Each administrator in a chat generates their own invite links. Bots can't use invite links generated by other administrators. If you want your bot to work with invite links, it will need to generate its own link using *aiogram.methods.export_chat_invite_link.ExportChatInviteLink* or by calling the *aiogram.methods.get_chat.GetChat* method. If your bot needs to generate a new primary invite link replacing its previous one, use *aiogram.methods.export_chat_invite_link.ExportChatInviteLink* again.

Source: https://core.telegram.org/bots/api#exportchatinvitelink

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: str = await bot.export_chat_invite_link(...)
```

Method as object

Imports:

- `from aiogram.methods.export_chat_invite_link import ExportChatInviteLink`
- alias: `from aiogram.methods import ExportChatInviteLink`

With specific bot

```
result: str = await bot(ExportChatInviteLink(...))
```

As reply into Webhook in handler

```
return ExportChatInviteLink(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.export_invite_link()*

forwardMessage

Returns: `Message`

class `aiogram.methods.forward_message.`**`ForwardMessage`**(**, chat_id: int | str, from_chat_id: int | str, message_id: int, message_thread_id: int | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, **extra_data: ~typing.Any*)

Use this method to forward messages of any kind. Service messages and messages with protected content can't be forwarded. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#forwardmessage

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

from_chat_id: int | str

Unique identifier for the chat where the original message was sent (or channel username in the format `@channelusername`)

message_id: int

Message identifier in the chat specified in *from_chat_id*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the forwarded message from forwarding and saving

Usage

As bot method

```
result: Message = await bot.forward_message(...)
```

Method as object

Imports:

- `from aiogram.methods.forward_message import ForwardMessage`
- alias: `from aiogram.methods import ForwardMessage`

With specific bot

```
result: Message = await bot(ForwardMessage(...))
```

As reply into Webhook in handler

```
return ForwardMessage(...)
```

As shortcut from received object

- `aiogram.types.message.Message.forward()`

forwardMessages

Returns: `List[MessageId]`

class `aiogram.methods.forward_messages.`**`ForwardMessages`**(**, chat_id: int | str, from_chat_id: int | str, message_ids: List[int], message_thread_id: int | None = None, disable_notification: bool | None = None, protect_content: bool | None = None, **extra_data: Any*)

Use this method to forward multiple messages of any kind. If some of the specified messages can't be found or forwarded, they are skipped. Service messages and messages with protected content can't be forwarded. Album grouping is kept for forwarded messages. On success, an array of `aiogram.types.message_id.MessageId` of the sent messages is returned.

Source: https://core.telegram.org/bots/api#forwardmessages

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`from_chat_id: int | str`

Unique identifier for the chat where the original messages were sent (or channel username in the format `@channelusername`)

`message_ids: List[int]`

A JSON-serialized list of 1-100 identifiers of messages in the chat *from_chat_id* to forward. The identifiers must be specified in a strictly increasing order.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`message_thread_id: int | None`

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

`disable_notification: bool | None`

Sends the messages silently. Users will receive a notification with no sound.

`protect_content: bool | None`

Protects the contents of the forwarded messages from forwarding and saving

Usage

As bot method

```
result: List[MessageId] = await bot.forward_messages(...)
```

Method as object

Imports:

- `from aiogram.methods.forward_messages import ForwardMessages`
- alias: `from aiogram.methods import ForwardMessages`

With specific bot

```
result: List[MessageId] = await bot(ForwardMessages(...))
```

As reply into Webhook in handler

```
return ForwardMessages(...)
```

getBusinessConnection

Returns: `BusinessConnection`

class aiogram.methods.get_business_connection.**GetBusinessConnection**(*, *business_connection_id: str*, ***extra_data: Any*)

Use this method to get information about the connection of the bot with a business account. Returns a *aiogram.types.business_connection.BusinessConnection* object on success.

Source: https://core.telegram.org/bots/api#getbusinessconnection

business_connection_id: str

Unique identifier of the business connection

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: BusinessConnection = await bot.get_business_connection(...)
```

Method as object

Imports:

- `from aiogram.methods.get_business_connection import GetBusinessConnection`
- alias: `from aiogram.methods import GetBusinessConnection`

With specific bot

```
result: BusinessConnection = await bot(GetBusinessConnection(...))
```

getChat

Returns: `ChatFullInfo`

class `aiogram.methods.get_chat.`**`GetChat`**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method to get up-to-date information about the chat. Returns a *`aiogram.types.chat_full_info.ChatFullInfo`* object on success.

Source: https://core.telegram.org/bots/api#getchat

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup or channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: ChatFullInfo = await bot.get_chat(...)
```

Method as object

Imports:

- `from aiogram.methods.get_chat import GetChat`
- alias: `from aiogram.methods import GetChat`

With specific bot

```
result: ChatFullInfo = await bot(GetChat(...))
```

getChatAdministrators

Returns: `List[Union[ChatMemberOwner, ChatMemberAdministrator, ChatMemberMember, ChatMemberRestricted, ChatMemberLeft, ChatMemberBanned]]`

class `aiogram.methods.get_chat_administrators.`**`GetChatAdministrators`**(**, chat_id: int | str, **extra_data: Any*)

Use this method to get a list of administrators in a chat, which aren't bots. Returns an Array of *`aiogram.types.chat_member.ChatMember`* objects.

Source: https://core.telegram.org/bots/api#getchatadministrators

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup or channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: List[Union[ChatMemberOwner, ChatMemberAdministrator, ChatMemberMember,␣
↪ChatMemberRestricted, ChatMemberLeft, ChatMemberBanned]] = await bot.get_chat_
↪administrators(...)
```

Method as object

Imports:

- `from aiogram.methods.get_chat_administrators import GetChatAdministrators`
- alias: `from aiogram.methods import GetChatAdministrators`

With specific bot

```
result: List[Union[ChatMemberOwner, ChatMemberAdministrator, ChatMemberMember,␣
↪ChatMemberRestricted, ChatMemberLeft, ChatMemberBanned]] = await␣
↪bot(GetChatAdministrators(...))
```

As shortcut from received object

- *`aiogram.types.chat.Chat.get_administrators()`*

getChatMember

Returns: `Union[ChatMemberOwner, ChatMemberAdministrator, ChatMemberMember, ChatMemberRestricted, ChatMemberLeft, ChatMemberBanned]`

class `aiogram.methods.get_chat_member.`**`GetChatMember`**(**, chat_id: int | str, user_id: int, **extra_data: Any*)

Use this method to get information about a member of a chat. The method is only guaranteed to work for other users if the bot is an administrator in the chat. Returns a *`aiogram.types.chat_member.ChatMember`* object on success.

Source: https://core.telegram.org/bots/api#getchatmember

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup or channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`user_id: int`

Unique identifier of the target user

Usage

As bot method

```
result: Union[ChatMemberOwner, ChatMemberAdministrator, ChatMemberMember,␣
↪ChatMemberRestricted, ChatMemberLeft, ChatMemberBanned] = await bot.get_chat_member(...
↪)
```

Method as object

Imports:

- `from aiogram.methods.get_chat_member import GetChatMember`
- alias: `from aiogram.methods import GetChatMember`

With specific bot

```
result: Union[ChatMemberOwner, ChatMemberAdministrator, ChatMemberMember,␣
↪ChatMemberRestricted, ChatMemberLeft, ChatMemberBanned] = await bot(GetChatMember(...))
```

As shortcut from received object

- *`aiogram.types.chat.Chat.get_member()`*

getChatMemberCount

Returns: `int`

`class aiogram.methods.get_chat_member_count.`**`GetChatMemberCount`**`(*, chat_id: int | str, **extra_data: Any)`

Use this method to get the number of members in a chat. Returns *Int* on success.

Source: https://core.telegram.org/bots/api#getchatmembercount

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup or channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init``(context: Any, /) → None`

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: int = await bot.get_chat_member_count(...)
```

Method as object

Imports:

- `from aiogram.methods.get_chat_member_count import GetChatMemberCount`
- alias: `from aiogram.methods import GetChatMemberCount`

With specific bot

```
result: int = await bot(GetChatMemberCount(...))
```

As shortcut from received object

- *`aiogram.types.chat.Chat.get_member_count()`*

getChatMenuButton

Returns: `Union[MenuButtonDefault, MenuButtonWebApp, MenuButtonCommands]`

class `aiogram.methods.get_chat_menu_button.`**`GetChatMenuButton`**(*, *chat_id: int | None = None*, ***extra_data: Any*)

Use this method to get the current value of the bot's menu button in a private chat, or the default menu button. Returns *`aiogram.types.menu_button.MenuButton`* on success.

Source: https://core.telegram.org/bots/api#getchatmenubutton

`chat_id: int | None`

Unique identifier for the target private chat. If not specified, default bot's menu button will be returned

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: Union[MenuButtonDefault, MenuButtonWebApp, MenuButtonCommands] = await bot.get_
↪chat_menu_button(...)
```

Method as object

Imports:

- `from aiogram.methods.get_chat_menu_button import GetChatMenuButton`
- alias: `from aiogram.methods import GetChatMenuButton`

With specific bot

```
result: Union[MenuButtonDefault, MenuButtonWebApp, MenuButtonCommands] = await␣
↪bot(GetChatMenuButton(...))
```

getFile

Returns: `File`

class `aiogram.methods.get_file.`**GetFile**(*, *file_id: str*, ***extra_data: Any*)

Use this method to get basic information about a file and prepare it for downloading. For the moment, bots can download files of up to 20MB in size. On success, a *aiogram.types.file.File* object is returned. The file can then be downloaded via the link `https://api.telegram.org/file/bot<token>/<file_path>`, where `<file_path>` is taken from the response. It is guaranteed that the link will be valid for at least 1 hour. When the link expires, a new one can be requested by calling *aiogram.methods.get_file.GetFile* again. **Note:** This function may not preserve the original file name and MIME type. You should save the file's MIME type and name (if available) when the File object is received.

Source: https://core.telegram.org/bots/api#getfile

file_id: str

File identifier to get information about

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: File = await bot.get_file(...)
```

Method as object

Imports:

- `from aiogram.methods.get_file import GetFile`
- alias: `from aiogram.methods import GetFile`

With specific bot

```
result: File = await bot(GetFile(...))
```

getForumTopicIconStickers

Returns: `List[Sticker]`

class `aiogram.methods.get_forum_topic_icon_stickers.`**`GetForumTopicIconStickers`**`(**`*extra_data: Any*`)`

Use this method to get custom emoji stickers, which can be used as a forum topic icon by any user. Requires no parameters. Returns an Array of *`aiogram.types.sticker.Sticker`* objects.

Source: https://core.telegram.org/bots/api#getforumtopiciconstickers

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init``(`*context: Any*, /`)` → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: List[Sticker] = await bot.get_forum_topic_icon_stickers(...)
```

Method as object

Imports:

- `from aiogram.methods.get_forum_topic_icon_stickers import GetForumTopicIconStickers`
- alias: `from aiogram.methods import GetForumTopicIconStickers`

With specific bot

```
result: List[Sticker] = await bot(GetForumTopicIconStickers(...))
```

getMe

Returns: `User`

class aiogram.methods.get_me.**GetMe**(***extra_data: Any*)

A simple method for testing your bot's authentication token. Requires no parameters. Returns basic information about the bot in form of a `aiogram.types.user.User` object.

Source: https://core.telegram.org/bots/api#getme

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: User = await bot.get_me(...)
```

Method as object

Imports:

- `from aiogram.methods.get_me import GetMe`
- alias: `from aiogram.methods import GetMe`

With specific bot

```
result: User = await bot(GetMe(...))
```

getMyCommands

Returns: `List[BotCommand]`

class aiogram.methods.get_my_commands.**GetMyCommands**(**, scope: BotCommandScopeDefault | BotCommandScopeAllPrivateChats | BotCommandScopeAllGroupChats | BotCommandScopeAllChatAdministrators | BotCommandScopeChat | BotCommandScopeChatAdministrators | BotCommandScopeChatMember | None = None, language_code: str | None = None, **extra_data: Any*)

Use this method to get the current list of the bot's commands for the given scope and user language. Returns an Array of *aiogram.types.bot_command.BotCommand* objects. If commands aren't set, an empty list is returned.

Source: https://core.telegram.org/bots/api#getmycommands

scope: *BotCommandScopeDefault* | *BotCommandScopeAllPrivateChats* | *BotCommandScopeAllGroupChats* | *BotCommandScopeAllChatAdministrators* | *BotCommandScopeChat* | *BotCommandScopeChatAdministrators* | *BotCommandScopeChatMember* | None

A JSON-serialized object, describing scope of users. Defaults to *aiogram.types.bot_command_scope_default.BotCommandScopeDefault*.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

language_code: str | None

A two-letter ISO 639-1 language code or an empty string

Usage

As bot method

```
result: List[BotCommand] = await bot.get_my_commands(...)
```

Method as object

Imports:

- `from aiogram.methods.get_my_commands import GetMyCommands`
- alias: `from aiogram.methods import GetMyCommands`

With specific bot

```
result: List[BotCommand] = await bot(GetMyCommands(...))
```

getMyDefaultAdministratorRights

Returns: `ChatAdministratorRights`

class `aiogram.methods.get_my_default_administrator_rights.`**`GetMyDefaultAdministratorRights`**(*, *for_channels: bool | None = None*, ***extra_data: Any*)

Use this method to get the current default administrator rights of the bot. Returns *aiogram.types.chat_administrator_rights.ChatAdministratorRights* on success.

Source: https://core.telegram.org/bots/api#getmydefaultadministratorrights

`for_channels: bool | None`

Pass `True` to get default administrator rights of the bot in channels. Otherwise, default administrator rights of the bot for groups and supergroups will be returned.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: ChatAdministratorRights = await bot.get_my_default_administrator_rights(...)
```

Method as object

Imports:

- `from aiogram.methods.get_my_default_administrator_rights import GetMyDefaultAdministratorRights`
- alias: `from aiogram.methods import GetMyDefaultAdministratorRights`

With specific bot

```
result: ChatAdministratorRights = await bot(GetMyDefaultAdministratorRights(...))
```

getMyDescription

Returns: `BotDescription`

class `aiogram.methods.get_my_description.`**`GetMyDescription`**(**, language_code: str | None = None, **extra_data: Any*)

Use this method to get the current bot description for the given user language. Returns *`aiogram.types.bot_description.BotDescription`* on success.

Source: https://core.telegram.org/bots/api#getmydescription

`language_code: str | None`

A two-letter ISO 639-1 language code or an empty string

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: BotDescription = await bot.get_my_description(...)
```

Method as object

Imports:

- `from aiogram.methods.get_my_description import GetMyDescription`
- alias: `from aiogram.methods import GetMyDescription`

With specific bot

```
result: BotDescription = await bot(GetMyDescription(...))
```

getMyName

Returns: `BotName`

class `aiogram.methods.get_my_name.`**GetMyName**(*, *language_code: str | None = None*, ***extra_data: Any*)

Use this method to get the current bot name for the given user language. Returns *aiogram.types.bot_name.BotName* on success.

Source: https://core.telegram.org/bots/api#getmyname

language_code: str | None

A two-letter ISO 639-1 language code or an empty string

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: BotName = await bot.get_my_name(...)
```

Method as object

Imports:

- `from aiogram.methods.get_my_name import GetMyName`
- alias: `from aiogram.methods import GetMyName`

With specific bot

```
result: BotName = await bot(GetMyName(...))
```

getMyShortDescription

Returns: `BotShortDescription`

class aiogram.methods.get_my_short_description.**GetMyShortDescription**(**, language_code: str | None = None, **extra_data: Any*)

Use this method to get the current bot short description for the given user language. Returns *aiogram.types.bot_short_description.BotShortDescription* on success.

Source: https://core.telegram.org/bots/api#getmyshortdescription

language_code: str | None

A two-letter ISO 639-1 language code or an empty string

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: BotShortDescription = await bot.get_my_short_description(...)
```

Method as object

Imports:

- `from aiogram.methods.get_my_short_description import GetMyShortDescription`
- alias: `from aiogram.methods import GetMyShortDescription`

With specific bot

```
result: BotShortDescription = await bot(GetMyShortDescription(...))
```

getUserChatBoosts

Returns: `UserChatBoosts`

class aiogram.methods.get_user_chat_boosts.**GetUserChatBoosts**(**, chat_id: int | str, user_id: int, **extra_data: Any*)

Use this method to get the list of boosts added to a chat by a user. Requires administrator rights in the chat. Returns a *aiogram.types.user_chat_boosts.UserChatBoosts* object.

Source: https://core.telegram.org/bots/api#getuserchatboosts

chat_id: int | str

Unique identifier for the chat or username of the channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

user_id: int

Unique identifier of the target user

Usage

As bot method

```
result: UserChatBoosts = await bot.get_user_chat_boosts(...)
```

Method as object

Imports:

- `from aiogram.methods.get_user_chat_boosts import GetUserChatBoosts`
- alias: `from aiogram.methods import GetUserChatBoosts`

With specific bot

```
result: UserChatBoosts = await bot(GetUserChatBoosts(...))
```

getUserProfilePhotos

Returns: `UserProfilePhotos`

class aiogram.methods.get_user_profile_photos.**GetUserProfilePhotos**(**, user_id: int, offset: int | None = None, limit: int | None = None, **extra_data: Any*)

Use this method to get a list of profile pictures for a user. Returns a *aiogram.types.user_profile_photos.UserProfilePhotos* object.

Source: https://core.telegram.org/bots/api#getuserprofilephotos

user_id: int

Unique identifier of the target user

offset: int | None

Sequential number of the first photo to be returned. By default, all photos are returned.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

limit: int | None

Limits the number of photos to be retrieved. Values between 1-100 are accepted. Defaults to 100.

Usage

As bot method

```
result: UserProfilePhotos = await bot.get_user_profile_photos(...)
```

Method as object

Imports:

- `from aiogram.methods.get_user_profile_photos import GetUserProfilePhotos`
- alias: `from aiogram.methods import GetUserProfilePhotos`

With specific bot

```
result: UserProfilePhotos = await bot(GetUserProfilePhotos(...))
```

As shortcut from received object

- *aiogram.types.user.User.get_profile_photos()*

hideGeneralForumTopic

Returns: `bool`

class `aiogram.methods.hide_general_forum_topic.`**HideGeneralForumTopic**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method to hide the 'General' topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights. The topic will be automatically closed if it was open. Returns `True` on success.

Source: https://core.telegram.org/bots/api#hidegeneralforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.hide_general_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.hide_general_forum_topic import HideGeneralForumTopic`
- alias: `from aiogram.methods import HideGeneralForumTopic`

With specific bot

```
result: bool = await bot(HideGeneralForumTopic(...))
```

As reply into Webhook in handler

```
return HideGeneralForumTopic(...)
```

leaveChat

Returns: `bool`

class `aiogram.methods.leave_chat.`**`LeaveChat`**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method for your bot to leave a group, supergroup or channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#leavechat

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup or channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.leave_chat(...)
```

Method as object

Imports:

- `from aiogram.methods.leave_chat import LeaveChat`
- alias: `from aiogram.methods import LeaveChat`

With specific bot

```
result: bool = await bot(LeaveChat(...))
```

As reply into Webhook in handler

```
return LeaveChat(...)
```

As shortcut from received object

- *`aiogram.types.chat.Chat.leave()`*

logOut

Returns: `bool`

class `aiogram.methods.log_out.`**`LogOut`**(***extra_data: Any*)

Use this method to log out from the cloud Bot API server before launching the bot locally. You **must** log out the bot before running it locally, otherwise there is no guarantee that the bot will receive updates. After a successful call, you can immediately log in on a local server, but will not be able to log in back to the cloud Bot API server for 10 minutes. Returns `True` on success. Requires no parameters.

Source: https://core.telegram.org/bots/api#logout

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.log_out(...)
```

Method as object

Imports:

- `from aiogram.methods.log_out import LogOut`
- alias: `from aiogram.methods import LogOut`

With specific bot

```
result: bool = await bot(LogOut(...))
```

As reply into Webhook in handler

```
return LogOut(...)
```

pinChatMessage

Returns: `bool`

class `aiogram.methods.pin_chat_message.`**`PinChatMessage`**(*, *chat_id: int* | *str*, *message_id: int*, *business_connection_id: str* | *None* = *None*, *disable_notification: bool* | *None* = *None*, ***extra_data: Any*)

Use this method to add a message to the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#pinchatmessage

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`message_id: int`

Identifier of a message to pin

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

business_connection_id: str | None
Unique identifier of the business connection on behalf of which the message will be pinned

disable_notification: bool | None
Pass `True` if it is not necessary to send a notification to all chat members about the new pinned message. Notifications are always disabled in channels and private chats.

Usage

As bot method

```
result: bool = await bot.pin_chat_message(...)
```

Method as object

Imports:

- `from aiogram.methods.pin_chat_message import PinChatMessage`
- alias: `from aiogram.methods import PinChatMessage`

With specific bot

```
result: bool = await bot(PinChatMessage(...))
```

As reply into Webhook in handler

```
return PinChatMessage(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.pin_message()*
- *aiogram.types.message.Message.pin()*

promoteChatMember

Returns: `bool`

`class aiogram.methods.promote_chat_member.PromoteChatMember`(**, chat_id: int | str, user_id: int, is_anonymous: bool | None = None, can_manage_chat: bool | None = None, can_delete_messages: bool | None = None, can_manage_video_chats: bool | None = None, can_restrict_members: bool | None = None, can_promote_members: bool | None = None, can_change_info: bool | None = None, can_invite_users: bool | None = None, can_post_stories: bool | None = None, can_edit_stories: bool | None = None, can_delete_stories: bool | None = None, can_post_messages: bool | None = None, can_edit_messages: bool | None = None, can_pin_messages: bool | None = None, can_manage_topics: bool | None = None, **extra_data: Any*)

Use this method to promote or demote a user in a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Pass `False` for all boolean parameters to demote a user. Returns `True` on success.

Source: https://core.telegram.org/bots/api#promotechatmember

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

user_id: int

Unique identifier of the target user

is_anonymous: bool | None

Pass `True` if the administrator's presence in the chat is hidden

can_manage_chat: bool | None

Pass `True` if the administrator can access the chat event log, get boost list, see hidden supergroup and channel members, report spam messages and ignore slow mode. Implied by any other administrator privilege.

can_delete_messages: bool | None

Pass `True` if the administrator can delete messages of other users

can_manage_video_chats: bool | None

Pass `True` if the administrator can manage video chats

can_restrict_members: bool | None

Pass `True` if the administrator can restrict, ban or unban chat members, or access supergroup statistics

can_promote_members: bool | None

Pass `True` if the administrator can add new administrators with a subset of their own privileges or demote administrators that they have promoted, directly or indirectly (promoted by administrators that were appointed by him)

can_change_info: bool | None

Pass `True` if the administrator can change chat title, photo and other settings

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

can_invite_users: bool | None

Pass `True` if the administrator can invite new users to the chat

can_post_stories: bool | None

Pass `True` if the administrator can post stories to the chat

can_edit_stories: bool | None

Pass `True` if the administrator can edit stories posted by other users, post stories to the chat page, pin chat stories, and access the chat's story archive

can_delete_stories: bool | None

Pass `True` if the administrator can delete stories posted by other users

can_post_messages: bool | None

Pass `True` if the administrator can post messages in the channel, or access channel statistics; for channels only

can_edit_messages: bool | None

Pass `True` if the administrator can edit messages of other users and can pin messages; for channels only

can_pin_messages: bool | None

Pass `True` if the administrator can pin messages; for supergroups only

can_manage_topics: bool | None

Pass `True` if the user is allowed to create, rename, close, and reopen forum topics; for supergroups only

Usage

As bot method

```
result: bool = await bot.promote_chat_member(...)
```

Method as object

Imports:

- `from aiogram.methods.promote_chat_member import PromoteChatMember`
- alias: `from aiogram.methods import PromoteChatMember`

With specific bot

```
result: bool = await bot(PromoteChatMember(...))
```

As reply into Webhook in handler

```
return PromoteChatMember(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.promote()*

reopenForumTopic

Returns: bool

class aiogram.methods.reopen_forum_topic.**ReopenForumTopic**(*, *chat_id: int | str*, *message_thread_id: int*, ***extra_data: Any*)

Use this method to reopen a closed topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights, unless it is the creator of the topic. Returns True on success.

Source: https://core.telegram.org/bots/api#reopenforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_thread_id: int

Unique identifier for the target message thread of the forum topic

Usage

As bot method

```
result: bool = await bot.reopen_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.reopen_forum_topic import ReopenForumTopic`
- alias: `from aiogram.methods import ReopenForumTopic`

With specific bot

```
result: bool = await bot(ReopenForumTopic(...))
```

As reply into Webhook in handler

```
return ReopenForumTopic(...)
```

reopenGeneralForumTopic

Returns: `bool`

class `aiogram.methods.reopen_general_forum_topic.`**`ReopenGeneralForumTopic`**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method to reopen a closed 'General' topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights. The topic will be automatically unhidden if it was hidden. Returns `True` on success.

Source: https://core.telegram.org/bots/api#reopengeneralforumtopic

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.reopen_general_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.reopen_general_forum_topic import ReopenGeneralForumTopic`
- alias: `from aiogram.methods import ReopenGeneralForumTopic`

With specific bot

```
result: bool = await bot(ReopenGeneralForumTopic(...))
```

As reply into Webhook in handler

```
return ReopenGeneralForumTopic(...)
```

restrictChatMember

Returns: `bool`

class `aiogram.methods.restrict_chat_member.`**`RestrictChatMember`**(*, *chat_id: int | str*, *user_id: int*, *permissions: ChatPermissions*, *use_independent_chat_permissions: bool | None = None*, *until_date: datetime | timedelta | int | None = None*, ***extra_data: Any*)

Use this method to restrict a user in a supergroup. The bot must be an administrator in the supergroup for this to work and must have the appropriate administrator rights. Pass `True` for all permissions to lift restrictions from a user. Returns `True` on success.

Source: https://core.telegram.org/bots/api#restrictchatmember

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

`user_id: int`

Unique identifier of the target user

`permissions:` ***`ChatPermissions`***

A JSON-serialized object for new user permissions

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`use_independent_chat_permissions: bool | None`

Pass `True` if chat permissions are set independently. Otherwise, the *can_send_other_messages* and *can_add_web_page_previews* permissions will imply the *can_send_messages*, *can_send_audios*, *can_send_documents*, *can_send_photos*, *can_send_videos*, *can_send_video_notes*, and *can_send_voice_notes* permissions; the *can_send_polls* permission will imply the *can_send_messages* permission.

until_date: datetime.datetime | datetime.timedelta | int | None

Date when restrictions will be lifted for the user; Unix time. If user is restricted for more than 366 days or less than 30 seconds from the current time, they are considered to be restricted forever

Usage

As bot method

```
result: bool = await bot.restrict_chat_member(...)
```

Method as object

Imports:

- `from aiogram.methods.restrict_chat_member import RestrictChatMember`
- alias: `from aiogram.methods import RestrictChatMember`

With specific bot

```
result: bool = await bot(RestrictChatMember(...))
```

As reply into Webhook in handler

```
return RestrictChatMember(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.restrict()*

revokeChatInviteLink

Returns: `ChatInviteLink`

class `aiogram.methods.revoke_chat_invite_link.`**RevokeChatInviteLink**(*, *chat_id: int | str*, *invite_link: str*, ***extra_data: Any*)

Use this method to revoke an invite link created by the bot. If the primary link is revoked, a new link is automatically generated. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns the revoked invite link as *aiogram.types.chat_invite_link.ChatInviteLink* object.

Source: https://core.telegram.org/bots/api#revokechatinvitelink

chat_id: int | str
Unique identifier of the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

invite_link: str
The invite link to revoke

Usage

As bot method

```
result: ChatInviteLink = await bot.revoke_chat_invite_link(...)
```

Method as object

Imports:

- `from aiogram.methods.revoke_chat_invite_link import RevokeChatInviteLink`
- alias: `from aiogram.methods import RevokeChatInviteLink`

With specific bot

```
result: ChatInviteLink = await bot(RevokeChatInviteLink(...))
```

As reply into Webhook in handler

```
return RevokeChatInviteLink(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.revoke_invite_link()*

sendAnimation

Returns: `Message`

class `aiogram.methods.send_animation.`**`SendAnimation`**(**, chat_id: int | str, animation: ~aiogram.types.input_file.InputFile | str, business_connection_id: str | None = None, message_thread_id: int | None = None, duration: int | None = None, width: int | None = None, height: int | None = None, thumbnail: ~aiogram.types.input_file.InputFile | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, has_spoiler: bool | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound). On success, the sent *`aiogram.types.message.Message`* is returned. Bots can currently send animation files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendanimation

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`animation:` *`InputFile`* **`| str`**

Animation to send. Pass a file_id as String to send an animation that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an animation from the Internet, or upload a new animation using multipart/form-data. *More information on Sending Files »*

`business_connection_id: str | None`

Unique identifier of the business connection on behalf of which the message will be sent

`message_thread_id: int | None`

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

duration: int | None

Duration of sent animation in seconds

width: int | None

Animation width

height: int | None

Animation height

thumbnail: *InputFile* | None

Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

caption: str | None

Animation caption (may also be used when resending animation by *file_id*), 0-1024 characters after entities parsing

parse_mode: str | Default | None

Mode for parsing entities in the animation caption. See formatting options for more details.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Pass `True`, if the caption must be shown above the message media

has_spoiler: bool | None

Pass `True` if the animation needs to be covered with a spoiler animation

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_animation(...)
```

Method as object

Imports:

- `from aiogram.methods.send_animation import SendAnimation`
- alias: `from aiogram.methods import SendAnimation`

With specific bot

```
result: Message = await bot(SendAnimation(...))
```

As reply into Webhook in handler

```
return SendAnimation(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_animation()*
- *aiogram.types.message.Message.reply_animation()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_animation()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_animation_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_animation()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_animation()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_animation()*

sendAudio

Returns: `Message`

class `aiogram.methods.send_audio.`**`SendAudio`**(*, *chat_id: int | str*, *audio: ~aiogram.types.input_file.InputFile | str*, *business_connection_id: str | None = None*, *message_thread_id: int | None = None*, *caption: str | None = None*, *parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>*, *caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *duration: int | None = None*, *performer: str | None = None*, *title: str | None = None*, *thumbnail: ~aiogram.types.input_file.InputFile | None = None*, *disable_notification: bool | None = None*, *protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>*, *message_effect_id: str | None = None*, *reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None*, *reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None*, *allow_sending_without_reply: bool | None = None*, *reply_to_message_id: int | None = None*, ***extra_data: ~typing.Any*)

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format. On success, the sent *`aiogram.types.message.Message`* is returned. Bots can currently send audio files of up to 50 MB in size, this limit may be changed in the future. For sending voice messages, use the *`aiogram.methods.send_voice.SendVoice`* method instead.

Source: https://core.telegram.org/bots/api#sendaudio

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`audio:` *`InputFile`* **`| str`**

Audio file to send. Pass a file_id as String to send an audio file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get an audio file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*

`business_connection_id: str | None`

Unique identifier of the business connection on behalf of which the message will be sent

`message_thread_id: int | None`

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

`caption: str | None`

Audio caption, 0-1024 characters after entities parsing

`parse_mode: str | Default | None`

Mode for parsing entities in the audio caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

duration: int | None

Duration of the audio in seconds

performer: str | None

Performer

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

title: str | None

Track name

thumbnail: *InputFile* | None

Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_audio(...)
```

Method as object

Imports:

- `from aiogram.methods.send_audio import SendAudio`
- alias: `from aiogram.methods import SendAudio`

With specific bot

```
result: Message = await bot(SendAudio(...))
```

As reply into Webhook in handler

```
return SendAudio(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_audio()*
- *aiogram.types.message.Message.reply_audio()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_audio()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_audio_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_audio()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_audio()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_audio()*

sendChatAction

Returns: `bool`

class `aiogram.methods.send_chat_action.`**`SendChatAction`**(**, chat_id: int | str, action: str, business_connection_id: str | None = None, message_thread_id: int | None = None, **extra_data: Any*)

Use this method when you need to tell the user that something is happening on the bot's side. The status is set for 5 seconds or less (when a message arrives from your bot, Telegram clients clear its typing status). Returns `True` on success.

Example: The ImageBot needs some time to process a request and upload the image. Instead of sending a text message along the lines of 'Retrieving image, please wait…', the bot may use `aiogram.methods.send_chat_action.SendChatAction` with *action = upload_photo*. The user will see a 'sending photo' status for the bot.

We only recommend using this method when a response from the bot will take a **noticeable** amount of time to arrive.

Source: https://core.telegram.org/bots/api#sendchataction

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

action: str

Type of action to broadcast. Choose one, depending on what the user is about to receive: *typing* for text messages, *upload_photo* for photos, *record_video* or *upload_video* for videos, *record_voice* or *upload_voice* for voice notes, *upload_document* for general files, *choose_sticker* for stickers, *find_location* for location data, *record_video_note* or *upload_video_note* for video notes.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the action will be sent

message_thread_id: int | None

Unique identifier for the target message thread; for supergroups only

Usage

As bot method

```
result: bool = await bot.send_chat_action(...)
```

Method as object

Imports:

- `from aiogram.methods.send_chat_action import SendChatAction`
- alias: `from aiogram.methods import SendChatAction`

With specific bot

```
result: bool = await bot(SendChatAction(...))
```

As reply into Webhook in handler

```
return SendChatAction(...)
```

As shortcut from received object

- `aiogram.types.chat.Chat.do()`

sendContact

Returns: `Message`

class `aiogram.methods.send_contact.`**`SendContact`**(**, chat_id: int | str, phone_number: str, first_name: str, business_connection_id: str | None = None, message_thread_id: int | None = None, last_name: str | None = None, vcard: str | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send phone contacts. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendcontact

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

phone_number: str

Contact's phone number

first_name: str

Contact's first name

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

last_name: str | None

Contact's last name

vcard: str | None

Additional data about the contact in the form of a vCard, 0-2048 bytes

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: ***ReplyParameters*** **| None**

Description of the message to reply to

reply_markup: ***InlineKeyboardMarkup*** **|** ***ReplyKeyboardMarkup*** **|** ***ReplyKeyboardRemove*** **|** ***ForceReply*** **| None**

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_contact(...)
```

Method as object

Imports:

- `from aiogram.methods.send_contact import SendContact`
- alias: `from aiogram.methods import SendContact`

With specific bot

```
result: Message = await bot(SendContact(...))
```

As reply into Webhook in handler

```
return SendContact(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_contact()*
- *aiogram.types.message.Message.reply_contact()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_contact()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_contact_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_contact()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_contact()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_contact()*

sendDice

Returns: `Message`

class aiogram.methods.send_dice.**SendDice**(**, chat_id: int | str, business_connection_id: str | None = None, message_thread_id: int | None = None, emoji: str | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send an animated emoji that will display a random value. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#senddice

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

emoji: str | None

Emoji on which the dice throw animation is based. Currently, must be one of '', '', '', '', '', or ''. Dice can have values 1-6 for '', '' and '', values 1-5 for '' and '', and values 1-64 for ''. Defaults to ''

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_dice(...)
```

Method as object

Imports:

- `from aiogram.methods.send_dice import SendDice`
- alias: `from aiogram.methods import SendDice`

With specific bot

```
result: Message = await bot(SendDice(...))
```

As reply into Webhook in handler

```
return SendDice(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_dice()*
- *aiogram.types.message.Message.reply_dice()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_dice()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_dice_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_dice()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_dice()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_dice()*

sendDocument

Returns: `Message`

class aiogram.methods.send_document.**SendDocument**(**, chat_id: int | str, document: ~aiogram.types.input_file.InputFile | str, business_connection_id: str | None = None, message_thread_id: int | None = None, thumbnail: ~aiogram.types.input_file.InputFile | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, disable_content_type_detection: bool | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send general files. On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send files of any type of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#senddocument

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

document: *InputFile* | str

File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

thumbnail: *InputFile* | None

Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

caption: str | None

Document caption (may also be used when resending documents by *file_id*), 0-1024 characters after entities parsing

parse_mode: str | Default | None

Mode for parsing entities in the document caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

disable_content_type_detection: bool | None

Disables automatic server-side content type detection for files uploaded using multipart/form-data

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_document(...)
```

Method as object

Imports:

- `from aiogram.methods.send_document import SendDocument`
- alias: `from aiogram.methods import SendDocument`

With specific bot

```
result: Message = await bot(SendDocument(...))
```

As reply into Webhook in handler

```
return SendDocument(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_document()*
- *aiogram.types.message.Message.reply_document()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_document()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_document_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_document()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_document()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_document()*

sendLocation

Returns: `Message`

class aiogram.methods.send_location.SendLocation(**, chat_id: int | str, latitude: float, longitude: float, business_connection_id: str | None = None, message_thread_id: int | None = None, horizontal_accuracy: float | None = None, live_period: int | None = None, heading: int | None = None, proximity_alert_radius: int | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send point on the map. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendlocation

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

latitude: float

Latitude of the location

longitude: float

Longitude of the location

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

horizontal_accuracy: float | None

The radius of uncertainty for the location, measured in meters; 0-1500

live_period: int | None

Period in seconds during which the location will be updated (see Live Locations, should be between 60 and 86400, or 0x7FFFFFFF for live locations that can be edited indefinitely.

heading: int | None

For live locations, a direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

proximity_alert_radius: int | None

For live locations, a maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_location(...)
```

Method as object

Imports:

- `from aiogram.methods.send_location import SendLocation`
- alias: `from aiogram.methods import SendLocation`

With specific bot

```
result: Message = await bot(SendLocation(...))
```

As reply into Webhook in handler

```
return SendLocation(...)
```

As shortcut from received object

- `aiogram.types.message.Message.answer_location()`
- `aiogram.types.message.Message.reply_location()`
- `aiogram.types.chat_join_request.ChatJoinRequest.answer_location()`
- `aiogram.types.chat_join_request.ChatJoinRequest.answer_location_pm()`
- `aiogram.types.chat_member_updated.ChatMemberUpdated.answer_location()`
- `aiogram.types.inaccessible_message.InaccessibleMessage.answer_location()`
- `aiogram.types.inaccessible_message.InaccessibleMessage.reply_location()`

sendMediaGroup

Returns: `List[Message]`

class `aiogram.methods.send_media_group.`**`SendMediaGroup`**(**, chat_id: int | str, media: ~typing.List[~aiogram.types.input_media_audio.InputMediaAudio | ~aiogram.types.input_media_document.InputMediaDocument | ~aiogram.types.input_media_photo.InputMediaPhoto | ~aiogram.types.input_media_video.InputMediaVideo], business_connection_id: str | None = None, message_thread_id: int | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send a group of photos, videos, documents or audios as an album. Documents and audio files can be only grouped in an album with messages of the same type. On success, an array of Messages that were sent is returned.

Source: https://core.telegram.org/bots/api#sendmediagroup

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

media: List[*InputMediaAudio* | *InputMediaDocument* | *InputMediaPhoto* | *InputMediaVideo*]

A JSON-serialized array describing messages to be sent, must include 2-10 items

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

disable_notification: bool | None

Sends messages silently. Users will receive a notification with no sound.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

protect_content: bool | Default | None

Protects the contents of the sent messages from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the messages are a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: List[Message] = await bot.send_media_group(...)
```

Method as object

Imports:

- `from aiogram.methods.send_media_group import SendMediaGroup`
- alias: `from aiogram.methods import SendMediaGroup`

With specific bot

```
result: List[Message] = await bot(SendMediaGroup(...))
```

As reply into Webhook in handler

```
return SendMediaGroup(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_media_group()*
- *aiogram.types.message.Message.reply_media_group()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_media_group()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_media_group_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_media_group()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_media_group()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_media_group()*

sendMessage

Returns: `Message`

class `aiogram.methods.send_message.`**`SendMessage`**(**, chat_id: int | str, text: str, business_connection_id: str | None = None, message_thread_id: int | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, link_preview_options: ~aiogram.types.link_preview_options.LinkPreviewOptions | ~aiogram.client.default.Default | None = <Default('link_preview')>, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, disable_web_page_preview: bool | ~aiogram.client.default.Default | None = <Default('link_preview_is_disabled')>, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send text messages. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendmessage

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`text: str`

Text of the message to be sent, 1-4096 characters after entities parsing

`business_connection_id: str | None`

Unique identifier of the business connection on behalf of which the message will be sent

`message_thread_id: int | None`

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

`parse_mode: str | Default | None`

Mode for parsing entities in the message text. See formatting options for more details.

`entities: List[MessageEntity] | None`

A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*

`link_preview_options: LinkPreviewOptions | Default | None`

Link preview generation options for the message

`disable_notification: bool | None`

Sends the message silently. Users will receive a notification with no sound.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* **| None**

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* **|** *ReplyKeyboardMarkup* **|** *ReplyKeyboardRemove* **|** *ForceReply* **| None**

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

disable_web_page_preview: bool | Default | None

Disables link previews for links in this message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_message(...)
```

Method as object

Imports:

- `from aiogram.methods.send_message import SendMessage`
- alias: `from aiogram.methods import SendMessage`

With specific bot

```
result: Message = await bot(SendMessage(...))
```

As reply into Webhook in handler

```
return SendMessage(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer()*
- *aiogram.types.message.Message.reply()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply()*

sendPaidMedia

Returns: `Message`

class aiogram.methods.send_paid_media.**SendPaidMedia**(**, chat_id: int | str, star_count: int, media: List[InputPaidMediaPhoto | InputPaidMediaVideo], business_connection_id: str | None = None, payload: str | None = None, caption: str | None = None, parse_mode: str | None = None, caption_entities: List[MessageEntity] | None = None, show_caption_above_media: bool | None = None, disable_notification: bool | None = None, protect_content: bool | None = None, reply_parameters: ReplyParameters | None = None, reply_markup: InlineKeyboardMarkup | ReplyKeyboardMarkup | ReplyKeyboardRemove | ForceReply | None = None, **extra_data: Any*)

Use this method to send paid media. On success, the sent *aiogram.types.message.Message* is returned.

Source: https://core.telegram.org/bots/api#sendpaidmedia

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`). If the chat is a channel, all Telegram Star proceeds from this media will be credited to the chat's balance. Otherwise, they will be credited to the bot's balance.

star_count: int

The number of Telegram Stars that must be paid to buy access to the media; 1-2500

media: List[*InputPaidMediaPhoto* | *InputPaidMediaVideo*]

A JSON-serialized array describing the media to be sent; up to 10 items

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

payload: str | None

Bot-defined paid media payload, 0-128 bytes. This will not be displayed to the user, use it for your internal processes.

caption: str | None

Media caption, 0-1024 characters after entities parsing

parse_mode: str | None

Mode for parsing entities in the media caption. See formatting options for more details.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | None

Pass `True`, if the caption must be shown above the message media

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | None

Protects the contents of the sent message from forwarding and saving

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

Usage

As bot method

```
result: Message = await bot.send_paid_media(...)
```

Method as object

Imports:

- `from aiogram.methods.send_paid_media import SendPaidMedia`
- alias: `from aiogram.methods import SendPaidMedia`

With specific bot

```
result: Message = await bot(SendPaidMedia(...))
```

As reply into Webhook in handler

```
return SendPaidMedia(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_paid_media()*
- *aiogram.types.message.Message.reply_paid_media()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_paid_media()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_paid_media()*

sendPhoto

Returns: `Message`

class aiogram.methods.send_photo.**SendPhoto**(**, chat_id: int | str, photo: ~aiogram.types.input_file.InputFile | str, business_connection_id: str | None = None, message_thread_id: int | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, has_spoiler: bool | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send photos. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendphoto

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

photo: *InputFile* | str

Photo to send. Pass a file_id as String to send a photo that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a photo from the Internet, or upload a new photo using multipart/form-data. The photo must be at most 10 MB in size. The photo's width and height must not exceed 10000 in total. Width and height ratio must be at most 20. *More information on Sending Files »*

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

caption: str | None

Photo caption (may also be used when resending photos by *file_id*), 0-1024 characters after entities parsing

parse_mode: str | Default | None

Mode for parsing entities in the photo caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Pass `True`, if the caption must be shown above the message media

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

has_spoiler: bool | None

Pass `True` if the photo needs to be covered with a spoiler animation

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: ***ReplyParameters*** **| None**

Description of the message to reply to

reply_markup: ***InlineKeyboardMarkup*** **|** ***ReplyKeyboardMarkup*** **|** ***ReplyKeyboardRemove*** **|** ***ForceReply*** **| None**

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_photo(...)
```

Method as object

Imports:

- `from aiogram.methods.send_photo import SendPhoto`
- alias: `from aiogram.methods import SendPhoto`

With specific bot

```
result: Message = await bot(SendPhoto(...))
```

As reply into Webhook in handler

```
return SendPhoto(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_photo()*
- *aiogram.types.message.Message.reply_photo()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_photo()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_photo_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_photo()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_photo()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_photo()*

sendPoll

Returns: `Message`

class aiogram.methods.send_poll.**SendPoll**(**, chat_id: int | str, question: str, options: ~typing.List[~aiogram.types.input_poll_option.InputPollOption | str], business_connection_id: str | None = None, message_thread_id: int | None = None, question_parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, question_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, is_anonymous: bool | None = None, type: str | None = None, allows_multiple_answers: bool | None = None, correct_option_id: int | None = None, explanation: str | None = None, explanation_parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, explanation_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, open_period: int | None = None, close_date: ~datetime.datetime | ~datetime.timedelta | int | None = None, is_closed: bool | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send a native poll. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendpoll

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

question: str

Poll question, 1-300 characters

options: List[*InputPollOption* | str]

A JSON-serialized list of 2-10 answer options

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

question_parse_mode: str | Default | None

Mode for parsing entities in the question. See formatting options for more details. Currently, only custom emoji entities are allowed

question_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the poll question. It can be specified instead of *question_parse_mode*

is_anonymous: bool | None

`True`, if the poll needs to be anonymous, defaults to `True`

type: str | None

Poll type, 'quiz' or 'regular', defaults to 'regular'

allows_multiple_answers: bool | None

`True`, if the poll allows multiple answers, ignored for polls in quiz mode, defaults to `False`

correct_option_id: int | None

0-based identifier of the correct answer option, required for polls in quiz mode

explanation: str | None

Text that is shown when a user chooses an incorrect answer or taps on the lamp icon in a quiz-style poll, 0-200 characters with at most 2 line feeds after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

explanation_parse_mode: str | Default | None

Mode for parsing entities in the explanation. See formatting options for more details.

explanation_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the poll explanation. It can be specified instead of *explanation_parse_mode*

open_period: int | None

Amount of time in seconds the poll will be active after creation, 5-600. Can't be used together with *close_date*.

close_date: datetime.datetime | datetime.timedelta | int | None

Point in time (Unix timestamp) when the poll will be automatically closed. Must be at least 5 and no more than 600 seconds in the future. Can't be used together with *open_period*.

is_closed: bool | None

Pass `True` if the poll needs to be immediately closed. This can be useful for poll preview.

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* **|** *ReplyKeyboardMarkup* **|** *ReplyKeyboardRemove* **|** *ForceReply* **| None**

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_poll(...)
```

Method as object

Imports:

- `from aiogram.methods.send_poll import SendPoll`
- alias: `from aiogram.methods import SendPoll`

With specific bot

```
result: Message = await bot(SendPoll(...))
```

As reply into Webhook in handler

```
return SendPoll(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_poll()*
- *aiogram.types.message.Message.reply_poll()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_poll()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_poll_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_poll()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_poll()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_poll()*

sendVenue

Returns: `Message`

class `aiogram.methods.send_venue.`**SendVenue**(*, *chat_id: int | str*, *latitude: float*, *longitude: float*, *title: str*, *address: str*, *business_connection_id: str | None = None*, *message_thread_id: int | None = None*, *foursquare_id: str | None = None*, *foursquare_type: str | None = None*, *google_place_id: str | None = None*, *google_place_type: str | None = None*, *disable_notification: bool | None = None*, *protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>*, *message_effect_id: str | None = None*, *reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None*, *reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None*, *allow_sending_without_reply: bool | None = None*, *reply_to_message_id: int | None = None*, ***extra_data: ~typing.Any*)

Use this method to send information about a venue. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendvenue

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

latitude: float

Latitude of the venue

longitude: float

Longitude of the venue

title: str

Name of the venue

address: str

Address of the venue

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

foursquare_id: str | None

Foursquare identifier of the venue

foursquare_type: str | None

Foursquare type of the venue, if known. (For example, ‘arts_entertainment/default’, ‘arts_entertainment/aquarium’ or ‘food/icecream’.)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

google_place_id: str | None

Google Places identifier of the venue

google_place_type: str | None

Google Places type of the venue. (See supported types.)

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: ***ReplyParameters*** **| None**

Description of the message to reply to

reply_markup: ***InlineKeyboardMarkup*** **|** ***ReplyKeyboardMarkup*** **|** ***ReplyKeyboardRemove*** **|** ***ForceReply*** **| None**

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_venue(...)
```

Method as object

Imports:

- `from aiogram.methods.send_venue import SendVenue`
- alias: `from aiogram.methods import SendVenue`

With specific bot

```
result: Message = await bot(SendVenue(...))
```

As reply into Webhook in handler

```
return SendVenue(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_venue()*
- *aiogram.types.message.Message.reply_venue()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_venue()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_venue_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_venue()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_venue()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_venue()*

sendVideo

Returns: `Message`

class aiogram.methods.send_video.**SendVideo**(**, chat_id: int | str, video: ~aiogram.types.input_file.InputFile | str, business_connection_id: str | None = None, message_thread_id: int | None = None, duration: int | None = None, width: int | None = None, height: int | None = None, thumbnail: ~aiogram.types.input_file.InputFile | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, has_spoiler: bool | None = None, supports_streaming: bool | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send video files, Telegram clients support MPEG4 videos (other formats may be sent as `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send video files of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvideo

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

video: *InputFile* | str

Video to send. Pass a file_id as String to send a video that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a video from the Internet, or upload a new video using multipart/form-data. *More information on Sending Files »*

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

duration: int | None

Duration of sent video in seconds

width: int | None

Video width

height: int | None

Video height

thumbnail: *InputFile* | None

Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

caption: str | None

Video caption (may also be used when resending videos by *file_id*), 0-1024 characters after entities parsing

parse_mode: str | Default | None

Mode for parsing entities in the video caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

show_caption_above_media: bool | Default | None

Pass `True`, if the caption must be shown above the message media

has_spoiler: bool | None

Pass `True` if the video needs to be covered with a spoiler animation

supports_streaming: bool | None

Pass `True` if the uploaded video is suitable for streaming

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_video(...)
```

Method as object

Imports:

- `from aiogram.methods.send_video import SendVideo`
- alias: `from aiogram.methods import SendVideo`

With specific bot

```
result: Message = await bot(SendVideo(...))
```

As reply into Webhook in handler

```
return SendVideo(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_video()*
- *aiogram.types.message.Message.reply_video()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_video()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_video_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_video()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_video()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_video()*

sendVideoNote

Returns: `Message`

class `aiogram.methods.send_video_note.`**SendVideoNote**(**, chat_id: int | str, video_note: ~aiogram.types.input_file.InputFile | str, business_connection_id: str | None = None, message_thread_id: int | None = None, duration: int | None = None, length: int | None = None, thumbnail: ~aiogram.types.input_file.InputFile | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

As of v.4.0, Telegram clients support rounded square MPEG4 videos of up to 1 minute long. Use this method to send video messages. On success, the sent *`aiogram.types.message.Message`* is returned.

Source: https://core.telegram.org/bots/api#sendvideonote

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`video_note:` ***`InputFile`*** **`| str`**

Video note to send. Pass a file_id as String to send a video note that exists on the Telegram servers (recommended) or upload a new video using multipart/form-data. *More information on Sending Files »*. Sending video notes by a URL is currently unsupported

`business_connection_id: str | None`

Unique identifier of the business connection on behalf of which the message will be sent

`message_thread_id: int | None`

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

`duration: int | None`

Duration of sent video in seconds

`length: int | None`

Video width and height, i.e. diameter of the video message

`thumbnail:` ***`InputFile`*** **`| None`**

Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_video_note(...)
```

Method as object

Imports:

- `from aiogram.methods.send_video_note import SendVideoNote`
- alias: `from aiogram.methods import SendVideoNote`

With specific bot

```
result: Message = await bot(SendVideoNote(...))
```

As reply into Webhook in handler

```
return SendVideoNote(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_video_note()*
- *aiogram.types.message.Message.reply_video_note()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_video_note()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_video_note_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_video_note()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_video_note()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_video_note()*

sendVoice

Returns: `Message`

class aiogram.methods.send_voice.**SendVoice**(**, chat_id: int | str, voice: ~aiogram.types.input_file.InputFile | str, business_connection_id: str | None = None, message_thread_id: int | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, duration: int | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | ~aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup | ~aiogram.types.reply_keyboard_remove.ReplyKeyboardRemove | ~aiogram.types.force_reply.ForceReply | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS, or in .MP3 format, or in .M4A format (other formats may be sent as `aiogram.types.audio.Audio` or `aiogram.types.document.Document`). On success, the sent `aiogram.types.message.Message` is returned. Bots can currently send voice messages of up to 50 MB in size, this limit may be changed in the future.

Source: https://core.telegram.org/bots/api#sendvoice

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

voice: *InputFile* | str

Audio file to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data. *More information on Sending Files »*

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

caption: str | None

Voice message caption, 0-1024 characters after entities parsing

parse_mode: str | Default | None

Mode for parsing entities in the voice message caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

duration: int | None

Duration of the voice message in seconds

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | *ReplyKeyboardMarkup* | *ReplyKeyboardRemove* | *ForceReply* | None

Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove a reply keyboard or to force a reply from the user

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_voice(...)
```

Method as object

Imports:

- `from aiogram.methods.send_voice import SendVoice`
- alias: `from aiogram.methods import SendVoice`

With specific bot

```
result: Message = await bot(SendVoice(...))
```

As reply into Webhook in handler

```
return SendVoice(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_voice()*
- *aiogram.types.message.Message.reply_voice()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_voice()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_voice_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_voice()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_voice()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_voice()*

setChatAdministratorCustomTitle

Returns: `bool`

class `aiogram.methods.set_chat_administrator_custom_title.`**`SetChatAdministratorCustomTitle`**(**, chat_id: int | str, user_id: int, custom_title: str, **extra_data: Any*)

Use this method to set a custom title for an administrator in a supergroup promoted by the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatadministratorcustomtitle

`chat_id: int | str`

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

`user_id: int`

Unique identifier of the target user

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`custom_title: str`

New custom title for the administrator; 0-16 characters, emoji are not allowed

Usage

As bot method

```
result: bool = await bot.set_chat_administrator_custom_title(...)
```

Method as object

Imports:

- from aiogram.methods.set_chat_administrator_custom_title import SetChatAdministratorCustomTitle
- alias: from aiogram.methods import SetChatAdministratorCustomTitle

With specific bot

```
result: bool = await bot(SetChatAdministratorCustomTitle(...))
```

As reply into Webhook in handler

```
return SetChatAdministratorCustomTitle(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.set_administrator_custom_title()*

setChatDescription

Returns: bool

class aiogram.methods.set_chat_description.**SetChatDescription**(*, *chat_id: int | str*, *description: str | None = None*, ***extra_data: Any*)

Use this method to change the description of a group, a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns True on success.

Source: https://core.telegram.org/bots/api#setchatdescription

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format @channelusername)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

description: str | None

New chat description, 0-255 characters

Usage

As bot method

```
result: bool = await bot.set_chat_description(...)
```

Method as object

Imports:

- `from aiogram.methods.set_chat_description import SetChatDescription`
- alias: `from aiogram.methods import SetChatDescription`

With specific bot

```
result: bool = await bot(SetChatDescription(...))
```

As reply into Webhook in handler

```
return SetChatDescription(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.set_description()*

setChatMenuButton

Returns: `bool`

class aiogram.methods.set_chat_menu_button.**SetChatMenuButton**(*, *chat_id: int | None = None*, *menu_button: MenuButtonCommands | MenuButtonWebApp | MenuButtonDefault | None = None*, ***extra_data: Any*)

Use this method to change the bot's menu button in a private chat, or the default menu button. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatmenubutton

chat_id: int | None

Unique identifier for the target private chat. If not specified, default bot's menu button will be changed

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

menu_button: *MenuButtonCommands* **|** *MenuButtonWebApp* **|** *MenuButtonDefault* **| None**

A JSON-serialized object for the bot's new menu button. Defaults to *aiogram.types.menu_button_default.MenuButtonDefault*

Usage

As bot method

```
result: bool = await bot.set_chat_menu_button(...)
```

Method as object

Imports:

- `from aiogram.methods.set_chat_menu_button import SetChatMenuButton`
- alias: `from aiogram.methods import SetChatMenuButton`

With specific bot

```
result: bool = await bot(SetChatMenuButton(...))
```

As reply into Webhook in handler

```
return SetChatMenuButton(...)
```

setChatPermissions

Returns: `bool`

class aiogram.methods.set_chat_permissions.**SetChatPermissions**(**, *chat_id: int | str*, *permissions: ChatPermissions*, *use_independent_chat_permissions: bool | None = None*, ***extra_data: Any*)

Use this method to set default chat permissions for all members. The bot must be an administrator in the group or a supergroup for this to work and must have the *can_restrict_members* administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatpermissions

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

permissions: *ChatPermissions*

A JSON-serialized object for new default chat permissions

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

use_independent_chat_permissions: bool | None

Pass `True` if chat permissions are set independently. Otherwise, the *can_send_other_messages* and *can_add_web_page_previews* permissions will imply the *can_send_messages*, *can_send_audios*, *can_send_documents*, *can_send_photos*, *can_send_videos*, *can_send_video_notes*, and *can_send_voice_notes* permissions; the *can_send_polls* permission will imply the *can_send_messages* permission.

Usage

As bot method

```
result: bool = await bot.set_chat_permissions(...)
```

Method as object

Imports:

- `from aiogram.methods.set_chat_permissions import SetChatPermissions`
- alias: `from aiogram.methods import SetChatPermissions`

With specific bot

```
result: bool = await bot(SetChatPermissions(...))
```

As reply into Webhook in handler

```
return SetChatPermissions(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.set_permissions()*

setChatPhoto

Returns: `bool`

class `aiogram.methods.set_chat_photo.`**`SetChatPhoto`**(**, chat_id: int | str, photo: InputFile, **extra_data: Any*)

Use this method to set a new profile photo for the chat. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatphoto

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`photo:` ***`InputFile`***

New chat photo, uploaded using multipart/form-data

Usage

As bot method

```
result: bool = await bot.set_chat_photo(...)
```

Method as object

Imports:

- `from aiogram.methods.set_chat_photo import SetChatPhoto`
- alias: `from aiogram.methods import SetChatPhoto`

With specific bot

```
result: bool = await bot(SetChatPhoto(...))
```

As shortcut from received object

- *aiogram.types.chat.Chat.set_photo()*

setChatStickerSet

Returns: `bool`

class aiogram.methods.set_chat_sticker_set.**SetChatStickerSet**(**, chat_id: int | str, sticker_set_name: str, **extra_data: Any*)

Use this method to set a new group sticker set for a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Use the field *can_set_sticker_set* optionally returned in *aiogram.methods.get_chat.GetChat* requests to check if the bot can use this method. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchatstickerset

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sticker_set_name: str

Name of the sticker set to be set as the group sticker set

Usage

As bot method

```
result: bool = await bot.set_chat_sticker_set(...)
```

Method as object

Imports:

- `from aiogram.methods.set_chat_sticker_set import SetChatStickerSet`
- alias: `from aiogram.methods import SetChatStickerSet`

With specific bot

```
result: bool = await bot(SetChatStickerSet(...))
```

As reply into Webhook in handler

```
return SetChatStickerSet(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.set_sticker_set()*

setChatTitle

Returns: `bool`

class `aiogram.methods.set_chat_title.`**SetChatTitle**(*, *chat_id: int | str*, *title: str*, ***extra_data: Any*)

Use this method to change the title of a chat. Titles can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setchattitle

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

title: str

New chat title, 1-128 characters

Usage

As bot method

```
result: bool = await bot.set_chat_title(...)
```

Method as object

Imports:

- `from aiogram.methods.set_chat_title import SetChatTitle`
- alias: `from aiogram.methods import SetChatTitle`

With specific bot

```
result: bool = await bot(SetChatTitle(...))
```

As reply into Webhook in handler

```
return SetChatTitle(...)
```

As shortcut from received object

- *`aiogram.types.chat.Chat.set_title()`*

setMessageReaction

Returns: `bool`

class `aiogram.methods.set_message_reaction.`**`SetMessageReaction`**(**, chat_id: int | str, message_id: int, reaction: List[*ReactionTypeEmoji | ReactionTypeCustomEmoji | ReactionTypePaid*] | None = None, is_big: bool | None = None, **extra_data: Any*)

Use this method to change the chosen reactions on a message. Service messages can't be reacted to. Automatically forwarded messages from a channel to its discussion group have the same available reactions as messages in the channel. Bots can't use paid reactions. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setmessagereaction

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`message_id: int`

Identifier of the target message. If the message belongs to a media group, the reaction is set to the first non-deleted message in the group instead.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

reaction: List[*ReactionTypeEmoji* | *ReactionTypeCustomEmoji* | *ReactionTypePaid*] | None

A JSON-serialized list of reaction types to set on the message. Currently, as non-premium users, bots can set up to one reaction per message. A custom emoji reaction can be used if it is either already present on the message or explicitly allowed by chat administrators. Paid reactions can't be used by bots.

is_big: bool | None

Pass `True` to set the reaction with a big animation

Usage

As bot method

```
result: bool = await bot.set_message_reaction(...)
```

Method as object

Imports:

- `from aiogram.methods.set_message_reaction import SetMessageReaction`
- alias: `from aiogram.methods import SetMessageReaction`

With specific bot

```
result: bool = await bot(SetMessageReaction(...))
```

As reply into Webhook in handler

```
return SetMessageReaction(...)
```

As shortcut from received object

- *aiogram.types.message.Message.react()*

setMyCommands

Returns: `bool`

class aiogram.methods.set_my_commands.**SetMyCommands**(*, *commands: List[*BotCommand*]*, *scope:* BotCommandScopeDefault | BotCommandScopeAllPrivateChats | BotCommandScopeAllGroupChats | BotCommandScopeAllChatAdministrators | BotCommandScopeChat | BotCommandScopeChatAdministrators | BotCommandScopeChatMember | *None* = *None*, *language_code: str* | *None* = *None*, ***extra_data: Any*)

Use this method to change the list of the bot's commands. See this manual for more details about bot commands. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setmycommands

commands: List[*BotCommand*]

A JSON-serialized list of bot commands to be set as the list of the bot's commands. At most 100 commands can be specified.

scope: *BotCommandScopeDefault* | *BotCommandScopeAllPrivateChats* | *BotCommandScopeAllGroupChats* | *BotCommandScopeAllChatAdministrators* | *BotCommandScopeChat* | *BotCommandScopeChatAdministrators* | *BotCommandScopeChatMember* | None

A JSON-serialized object, describing scope of users for which the commands are relevant. Defaults to *aiogram.types.bot_command_scope_default.BotCommandScopeDefault*.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

language_code: str | None

A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

Usage

As bot method

```
result: bool = await bot.set_my_commands(...)
```

Method as object

Imports:

- `from aiogram.methods.set_my_commands import SetMyCommands`
- alias: `from aiogram.methods import SetMyCommands`

With specific bot

```
result: bool = await bot(SetMyCommands(...))
```

As reply into Webhook in handler

```
return SetMyCommands(...)
```

setMyDefaultAdministratorRights

Returns: `bool`

class `aiogram.methods.set_my_default_administrator_rights.`**`SetMyDefaultAdministratorRights`**(**, rights: ChatAdministratorRights | None = None, for_channels: bool | None = None, **extra_data: Any*)

Use this method to change the default administrator rights requested by the bot when it's added as an administrator to groups or channels. These rights will be suggested to users, but they are free to modify the list before adding the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setmydefaultadministratorrights

`rights:` *`ChatAdministratorRights`* **`| None`**

A JSON-serialized object describing new default administrator rights. If not specified, the default administrator rights will be cleared.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`for_channels: bool | None`

Pass `True` to change the default administrator rights of the bot in channels. Otherwise, the default administrator rights of the bot for groups and supergroups will be changed.

Usage

As bot method

```
result: bool = await bot.set_my_default_administrator_rights(...)
```

Method as object

Imports:

- `from aiogram.methods.set_my_default_administrator_rights import SetMyDefaultAdministratorRights`
- alias: `from aiogram.methods import SetMyDefaultAdministratorRights`

With specific bot

```
result: bool = await bot(SetMyDefaultAdministratorRights(...))
```

As reply into Webhook in handler

```
return SetMyDefaultAdministratorRights(...)
```

setMyDescription

Returns: `bool`

class `aiogram.methods.set_my_description.`**`SetMyDescription`**(**, description: str | None = None, language_code: str | None = None, **extra_data: Any*)

Use this method to change the bot's description, which is shown in the chat with the bot if the chat is empty. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setmydescription

`description: str | None`

New bot description; 0-512 characters. Pass an empty string to remove the dedicated description for the given language.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`language_code: str | None`

A two-letter ISO 639-1 language code. If empty, the description will be applied to all users for whose language there is no dedicated description.

Usage

As bot method

```
result: bool = await bot.set_my_description(...)
```

Method as object

Imports:

- `from aiogram.methods.set_my_description import SetMyDescription`
- alias: `from aiogram.methods import SetMyDescription`

With specific bot

```
result: bool = await bot(SetMyDescription(...))
```

As reply into Webhook in handler

```
return SetMyDescription(...)
```

setMyName

Returns: `bool`

class `aiogram.methods.set_my_name.`**`SetMyName`**(*, *name: str | None = None*, *language_code: str | None = None*, ***extra_data: Any*)

Use this method to change the bot's name. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setmyname

name: str | None

New bot name; 0-64 characters. Pass an empty string to remove the dedicated name for the given language.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

language_code: str | None

A two-letter ISO 639-1 language code. If empty, the name will be shown to all users for whose language there is no dedicated name.

Usage

As bot method

```
result: bool = await bot.set_my_name(...)
```

Method as object

Imports:

- `from aiogram.methods.set_my_name import SetMyName`
- alias: `from aiogram.methods import SetMyName`

With specific bot

```
result: bool = await bot(SetMyName(...))
```

As reply into Webhook in handler

```
return SetMyName(...)
```

setMyShortDescription

Returns: `bool`

class `aiogram.methods.set_my_short_description.`**`SetMyShortDescription`**(*, *short_description: str | None = None*, *language_code: str | None = None*, ***extra_data: Any*)

Use this method to change the bot's short description, which is shown on the bot's profile page and is sent together with the link when users share the bot. Returns `True` on success.

Source: https://core.telegram.org/bots/api#setmyshortdescription

`short_description: str | None`

New short description for the bot; 0-120 characters. Pass an empty string to remove the dedicated short description for the given language.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`language_code: str | None`

A two-letter ISO 639-1 language code. If empty, the short description will be applied to all users for whose language there is no dedicated short description.

Usage

As bot method

```
result: bool = await bot.set_my_short_description(...)
```

Method as object

Imports:

- `from aiogram.methods.set_my_short_description import SetMyShortDescription`
- alias: `from aiogram.methods import SetMyShortDescription`

With specific bot

```
result: bool = await bot(SetMyShortDescription(...))
```

As reply into Webhook in handler

```
return SetMyShortDescription(...)
```

unbanChatMember

Returns: `bool`

class `aiogram.methods.unban_chat_member.`**`UnbanChatMember`**(**, chat_id: int | str, user_id: int, only_if_banned: bool | None = None, **extra_data: Any*)

Use this method to unban a previously banned user in a supergroup or channel. The user will **not** return to the group or channel automatically, but will be able to join via link, etc. The bot must be an administrator for this to work. By default, this method guarantees that after the call the user is not a member of the chat, but will be able to join it. So if the user is a member of the chat they will also be **removed** from the chat. If you don't want this, use the parameter *only_if_banned*. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unbanchatmember

chat_id: int | str

Unique identifier for the target group or username of the target supergroup or channel (in the format `@channelusername`)

user_id: int

Unique identifier of the target user

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

only_if_banned: bool | None
Do nothing if the user is not banned

Usage

As bot method

```
result: bool = await bot.unban_chat_member(...)
```

Method as object

Imports:

- `from aiogram.methods.unban_chat_member import UnbanChatMember`
- alias: `from aiogram.methods import UnbanChatMember`

With specific bot

```
result: bool = await bot(UnbanChatMember(...))
```

As reply into Webhook in handler

```
return UnbanChatMember(...)
```

As shortcut from received object

- *`aiogram.types.chat.Chat.unban()`*

unbanChatSenderChat

Returns: `bool`

class `aiogram.methods.unban_chat_sender_chat.`**`UnbanChatSenderChat`**(*, *chat_id: int | str*, *sender_chat_id: int*, ***extra_data: Any*)

Use this method to unban a previously banned channel chat in a supergroup or channel. The bot must be an administrator for this to work and must have the appropriate administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unbanchatsenderchat

chat_id: int | str
Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

sender_chat_id: int

Unique identifier of the target sender chat

Usage

As bot method

```
result: bool = await bot.unban_chat_sender_chat(...)
```

Method as object

Imports:

- `from aiogram.methods.unban_chat_sender_chat import UnbanChatSenderChat`
- alias: `from aiogram.methods import UnbanChatSenderChat`

With specific bot

```
result: bool = await bot(UnbanChatSenderChat(...))
```

As reply into Webhook in handler

```
return UnbanChatSenderChat(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.unban_sender_chat()*

unhideGeneralForumTopic

Returns: `bool`

class aiogram.methods.unhide_general_forum_topic.**UnhideGeneralForumTopic**(**, chat_id: int | str, **extra_data: Any*)

Use this method to unhide the 'General' topic in a forum supergroup chat. The bot must be an administrator in the chat for this to work and must have the *can_manage_topics* administrator rights. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unhidegeneralforumtopic

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.unhide_general_forum_topic(...)
```

Method as object

Imports:

- `from aiogram.methods.unhide_general_forum_topic import UnhideGeneralForumTopic`
- alias: `from aiogram.methods import UnhideGeneralForumTopic`

With specific bot

```
result: bool = await bot(UnhideGeneralForumTopic(...))
```

As reply into Webhook in handler

```
return UnhideGeneralForumTopic(...)
```

unpinAllChatMessages

Returns: `bool`

class aiogram.methods.unpin_all_chat_messages.**UnpinAllChatMessages**(*, *chat_id: int | str*, ***extra_data: Any*)

Use this method to clear the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unpinallchatmessages

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.unpin_all_chat_messages(...)
```

Method as object

Imports:

- `from aiogram.methods.unpin_all_chat_messages import UnpinAllChatMessages`
- alias: `from aiogram.methods import UnpinAllChatMessages`

With specific bot

```
result: bool = await bot(UnpinAllChatMessages(...))
```

As reply into Webhook in handler

```
return UnpinAllChatMessages(...)
```

As shortcut from received object

- *`aiogram.types.chat.Chat.unpin_all_messages()`*

unpinAllForumTopicMessages

Returns: `bool`

class `aiogram.methods.unpin_all_forum_topic_messages.`**`UnpinAllForumTopicMessages`**(*, *chat_id: int | str*, *message_thread_id: int*, ***extra_data: Any*)

Use this method to clear the list of pinned messages in a forum topic. The bot must be an administrator in the chat for this to work and must have the *can_pin_messages* administrator right in the supergroup. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unpinallforumtopicmessages

chat_id: int | str

Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

message_thread_id: int
Unique identifier for the target message thread of the forum topic

Usage

As bot method

```
result: bool = await bot.unpin_all_forum_topic_messages(...)
```

Method as object

Imports:

- `from aiogram.methods.unpin_all_forum_topic_messages import UnpinAllForumTopicMessages`
- alias: `from aiogram.methods import UnpinAllForumTopicMessages`

With specific bot

```
result: bool = await bot(UnpinAllForumTopicMessages(...))
```

As reply into Webhook in handler

```
return UnpinAllForumTopicMessages(...)
```

unpinAllGeneralForumTopicMessages

Returns: `bool`

class aiogram.methods.unpin_all_general_forum_topic_messages.**UnpinAllGeneralForumTopicMessages**(**, chat_id: int | str, **extra_data: Any*)

Use this method to clear the list of pinned messages in a General forum topic. The bot must be an administrator in the chat for this to work and must have the *can_pin_messages* administrator right in the supergroup. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unpinallgeneralforumtopicmessages

chat_id: int | str
Unique identifier for the target chat or username of the target supergroup (in the format `@supergroupusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.unpin_all_general_forum_topic_messages(...)
```

Method as object

Imports:

- `from aiogram.methods.unpin_all_general_forum_topic_messages import UnpinAllGeneralForumTopicMessages`
- alias: `from aiogram.methods import UnpinAllGeneralForumTopicMessages`

With specific bot

```
result: bool = await bot(UnpinAllGeneralForumTopicMessages(...))
```

As reply into Webhook in handler

```
return UnpinAllGeneralForumTopicMessages(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.unpin_all_general_forum_topic_messages()*

unpinChatMessage

Returns: `bool`

class aiogram.methods.unpin_chat_message.**UnpinChatMessage**(**, chat_id: int | str, business_connection_id: str | None = None, message_id: int | None = None, **extra_data: Any*)

Use this method to remove a message from the list of pinned messages in a chat. If the chat is not a private chat, the bot must be an administrator in the chat for this to work and must have the 'can_pin_messages' administrator right in a supergroup or 'can_edit_messages' administrator right in a channel. Returns `True` on success.

Source: https://core.telegram.org/bots/api#unpinchatmessage

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`business_connection_id: str | None`

Unique identifier of the business connection on behalf of which the message will be unpinned

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`message_id: int | None`

Identifier of the message to unpin. Required if *business_connection_id* is specified. If not specified, the most recent pinned message (by sending date) will be unpinned.

Usage

As bot method

```
result: bool = await bot.unpin_chat_message(...)
```

Method as object

Imports:

- `from aiogram.methods.unpin_chat_message import UnpinChatMessage`
- alias: `from aiogram.methods import UnpinChatMessage`

With specific bot

```
result: bool = await bot(UnpinChatMessage(...))
```

As reply into Webhook in handler

```
return UnpinChatMessage(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.unpin_message()*
- *aiogram.types.message.Message.unpin()*

Updating messages

deleteMessage

Returns: `bool`

class `aiogram.methods.delete_message.`**`DeleteMessage`**(*, *chat_id: int | str*, *message_id: int*, ***extra_data: Any*)

Use this method to delete a message, including service messages, with the following limitations:

- A message can only be deleted if it was sent less than 48 hours ago.
- Service messages about a supergroup, channel, or forum topic creation can't be deleted.
- A dice message in a private chat can only be deleted if it was sent more than 24 hours ago.
- Bots can delete outgoing messages in private chats, groups, and supergroups.
- Bots can delete incoming messages in private chats.
- Bots granted *can_post_messages* permissions can delete outgoing messages in channels.
- If the bot is an administrator of a group, it can delete any message there.
- If the bot has *can_delete_messages* permission in a supergroup or a channel, it can delete any message there.

Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletemessage

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`message_id: int`

Identifier of the message to delete

Usage

As bot method

```
result: bool = await bot.delete_message(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_message import DeleteMessage`
- alias: `from aiogram.methods import DeleteMessage`

With specific bot

```
result: bool = await bot(DeleteMessage(...))
```

As reply into Webhook in handler

```
return DeleteMessage(...)
```

As shortcut from received object

- *aiogram.types.chat.Chat.delete_message()*
- *aiogram.types.message.Message.delete()*

deleteMessages

Returns: `bool`

`class aiogram.methods.delete_messages.`**`DeleteMessages`**`(*, chat_id: int | str, message_ids: List[int], **extra_data: Any)`

Use this method to delete multiple messages simultaneously. If some of the specified messages can't be found, they are skipped. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletemessages

`chat_id: int | str`

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init``(context: Any, /)` → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`message_ids: List[int]`

A JSON-serialized list of 1-100 identifiers of messages to delete. See *aiogram.methods.delete_message.DeleteMessage* for limitations on which messages can be deleted

Usage

As bot method

```
result: bool = await bot.delete_messages(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_messages import DeleteMessages`
- alias: `from aiogram.methods import DeleteMessages`

With specific bot

```
result: bool = await bot(DeleteMessages(...))
```

As reply into Webhook in handler

```
return DeleteMessages(...)
```

editMessageCaption

Returns: `Union[Message, bool]`

class `aiogram.methods.edit_message_caption.`**`EditMessageCaption`**(**, business_connection_id: str | None = None, chat_id: int | str | None = None, message_id: int | None = None, inline_message_id: str | None = None, caption: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntit | None = None, show_caption_above_media: bool | ~aiogram.client.default.Default | None = <Default('show_caption_above_media')>, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeybo | None = None, **extra_data: ~typing.Any*)

Use this method to edit captions of messages. On success, if the edited message is not an inline message, the edited *aiogram.types.message.Message* is returned, otherwise True is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagecaption

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message to be edited was sent

chat_id: int | str | None

Required if *inline_message_id* is not specified. Unique identifier for the target chat or username of the target channel (in the format @channelusername)

message_id: int | None

Required if *inline_message_id* is not specified. Identifier of the message to edit

inline_message_id: str | None

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

caption: str | None

New caption of the message, 0-1024 characters after entities parsing

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Mode for parsing entities in the message caption. See formatting options for more details.

caption_entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in the caption, which can be specified instead of *parse_mode*

show_caption_above_media: bool | Default | None

Pass True, if the caption must be shown above the message media. Supported only for animation, photo and video messages.

reply_markup: *InlineKeyboardMarkup* | None

A JSON-serialized object for an inline keyboard.

Usage

As bot method

```
result: Union[Message, bool] = await bot.edit_message_caption(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_message_caption import EditMessageCaption`
- alias: `from aiogram.methods import EditMessageCaption`

With specific bot

```
result: Union[Message, bool] = await bot(EditMessageCaption(...))
```

As reply into Webhook in handler

```
return EditMessageCaption(...)
```

As shortcut from received object

- *aiogram.types.message.Message.edit_caption()*

editMessageLiveLocation

Returns: `Union[Message, bool]`

class aiogram.methods.edit_message_live_location.**EditMessageLiveLocation**(**, latitude: float, longitude: float, business_connection_id: str | None = None, chat_id: int | str | None = None, message_id: int | None = None, inline_message_id: str | None = None, live_period: int | None = None, horizontal_accuracy: float | None = None, heading: int | None = None, proximity_alert_radius: int | None = None, reply_markup: InlineKeyboardMarkup | None = None, **extra_data: Any*)

Use this method to edit live location messages. A location can be edited until its *live_period* expires or editing is explicitly disabled by a call to `aiogram.methods.stop_message_live_location.StopMessageLiveLocation`. On success, if the edited message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned.

Source: https://core.telegram.org/bots/api#editmessagelivelocation

latitude: float

Latitude of new location

longitude: float

Longitude of new location

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message to be edited was sent

chat_id: int | str | None

Required if *inline_message_id* is not specified. Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

message_id: int | None

Required if *inline_message_id* is not specified. Identifier of the message to edit

inline_message_id: str | None

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

live_period: int | None

New period in seconds during which the location can be updated, starting from the message send date. If 0x7FFFFFFF is specified, then the location can be updated forever. Otherwise, the new value must not exceed the current *live_period* by more than a day, and the live location expiration date must remain within the next 90 days. If not specified, then *live_period* remains unchanged

horizontal_accuracy: float | None

The radius of uncertainty for the location, measured in meters; 0-1500

heading: int | None

Direction in which the user is moving, in degrees. Must be between 1 and 360 if specified.

proximity_alert_radius: int | None

The maximum distance for proximity alerts about approaching another chat member, in meters. Must be between 1 and 100000 if specified.

reply_markup: *InlineKeyboardMarkup* | None

A JSON-serialized object for a new inline keyboard.

Usage

As bot method

```
result: Union[Message, bool] = await bot.edit_message_live_location(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_message_live_location import EditMessageLiveLocation`
- alias: `from aiogram.methods import EditMessageLiveLocation`

With specific bot

```
result: Union[Message, bool] = await bot(EditMessageLiveLocation(...))
```

As reply into Webhook in handler

```
return EditMessageLiveLocation(...)
```

As shortcut from received object

- *`aiogram.types.message.Message.edit_live_location()`*

editMessageMedia

Returns: `Union[Message, bool]`

class `aiogram.methods.edit_message_media.`**`EditMessageMedia`**(**, media: InputMediaAnimation | InputMediaDocument | InputMediaAudio | InputMediaPhoto | InputMediaVideo, business_connection_id: str | None = None, chat_id: int | str | None = None, message_id: int | None = None, inline_message_id: str | None = None, reply_markup: InlineKeyboardMarkup | None = None, **extra_data: Any*)

Use this method to edit animation, audio, document, photo, or video messages. If a message is part of a message album, then it can be edited only to an audio for audio albums, only to a document for document albums and to a photo or a video otherwise. When an inline message is edited, a new file can't be uploaded; use a previously uploaded file via its file_id or specify a URL. On success, if the edited message is not an inline message, the edited *`aiogram.types.message.Message`* is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagemedia

media: *InputMediaAnimation* **|** *InputMediaDocument* **|** *InputMediaAudio* **|** *InputMediaPhoto* **|** *InputMediaVideo*

A JSON-serialized object for a new media content of the message

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message to be edited was sent

chat_id: int | str | None

Required if *inline_message_id* is not specified. Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_id: int | None

Required if *inline_message_id* is not specified. Identifier of the message to edit

inline_message_id: str | None

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

reply_markup: *InlineKeyboardMarkup* **| None**

A JSON-serialized object for a new inline keyboard.

Usage

As bot method

```
result: Union[Message, bool] = await bot.edit_message_media(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_message_media import EditMessageMedia`
- alias: `from aiogram.methods import EditMessageMedia`

With specific bot

```
result: Union[Message, bool] = await bot(EditMessageMedia(...))
```

As reply into Webhook in handler

```
return EditMessageMedia(...)
```

As shortcut from received object

- *aiogram.types.message.Message.edit_media()*

editMessageReplyMarkup

Returns: `Union[Message, bool]`

class `aiogram.methods.edit_message_reply_markup.`**`EditMessageReplyMarkup`**(**, business_connection_id: str | None = None, chat_id: int | str | None = None, message_id: int | None = None, inline_message_id: str | None = None, reply_markup: InlineKeyboardMarkup | None = None, **extra_data: Any*)

Use this method to edit only the reply markup of messages. On success, if the edited message is not an inline message, the edited *aiogram.types.message.Message* is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagereplymarkup

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message to be edited was sent

chat_id: int | str | None

Required if *inline_message_id* is not specified. Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

message_id: int | None

Required if *inline_message_id* is not specified. Identifier of the message to edit

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

inline_message_id: str | None

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

reply_markup: ***InlineKeyboardMarkup*** **| None**

A JSON-serialized object for an inline keyboard.

Usage

As bot method

```
result: Union[Message, bool] = await bot.edit_message_reply_markup(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_message_reply_markup import EditMessageReplyMarkup`
- alias: `from aiogram.methods import EditMessageReplyMarkup`

With specific bot

```
result: Union[Message, bool] = await bot(EditMessageReplyMarkup(...))
```

As reply into Webhook in handler

```
return EditMessageReplyMarkup(...)
```

As shortcut from received object

- *aiogram.types.message.Message.edit_reply_markup()*
- *aiogram.types.message.Message.delete_reply_markup()*

editMessageText

Returns: `Union[Message, bool]`

class aiogram.methods.edit_message_text.**EditMessageText**(**, text: str, business_connection_id: str | None = None, chat_id: int | str | None = None, message_id: int | None = None, inline_message_id: str | None = None, parse_mode: str | ~aiogram.client.default.Default | None = <Default('parse_mode')>, entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None, link_preview_options: ~aiogram.types.link_preview_options.LinkPreviewOptions | ~aiogram.client.default.Default | None = <Default('link_preview')>, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMar | None = None, disable_web_page_preview: bool | ~aiogram.client.default.Default | None = <Default('link_preview_is_disabled')>, **extra_data: ~typing.Any*)

Use this method to edit text and game messages. On success, if the edited message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned. Note that business messages that were not sent by the bot and do not contain an inline keyboard can only be edited within **48 hours** from the time they were sent.

Source: https://core.telegram.org/bots/api#editmessagetext

text: str

New text of the message, 1-4096 characters after entities parsing

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message to be edited was sent

chat_id: int | str | None

Required if *inline_message_id* is not specified. Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

message_id: int | None

Required if *inline_message_id* is not specified. Identifier of the message to edit

inline_message_id: str | None

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

parse_mode: str | Default | None

Mode for parsing entities in the message text. See formatting options for more details.

entities: List[*MessageEntity*] | None

A JSON-serialized list of special entities that appear in message text, which can be specified instead of *parse_mode*

link_preview_options: *LinkPreviewOptions* | Default | None

Link preview generation options for the message

reply_markup: ***InlineKeyboardMarkup*** **| None**

A JSON-serialized object for an inline keyboard.

disable_web_page_preview: bool | Default | None

Disables link previews for links in this message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Union[Message, bool] = await bot.edit_message_text(...)
```

Method as object

Imports:

- `from aiogram.methods.edit_message_text import EditMessageText`
- alias: `from aiogram.methods import EditMessageText`

With specific bot

```
result: Union[Message, bool] = await bot(EditMessageText(...))
```

As reply into Webhook in handler

```
return EditMessageText(...)
```

As shortcut from received object

- *aiogram.types.message.Message.edit_text()*

stopMessageLiveLocation

Returns: `Union[Message, bool]`

class aiogram.methods.stop_message_live_location.**StopMessageLiveLocation**(*, *business_connection_id: str | None = None*, *chat_id: int | str | None = None*, *message_id: int | None = None*, *inline_message_id: str | None = None*, *reply_markup: InlineKeyboardMarkup | None = None*, ***extra_data: Any*)

Use this method to stop updating a live location message before *live_period* expires. On success, if the message is not an inline message, the edited `aiogram.types.message.Message` is returned, otherwise `True` is returned.

Source: https://core.telegram.org/bots/api#stopmessagelivelocation

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message to be edited was sent

chat_id: int | str | None

Required if *inline_message_id* is not specified. Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

message_id: int | None

Required if *inline_message_id* is not specified. Identifier of the message with live location to stop

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

inline_message_id: str | None

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

reply_markup: *InlineKeyboardMarkup* | None

A JSON-serialized object for a new inline keyboard.

Usage

As bot method

```
result: Union[Message, bool] = await bot.stop_message_live_location(...)
```

Method as object

Imports:

- `from aiogram.methods.stop_message_live_location import StopMessageLiveLocation`
- alias: `from aiogram.methods import StopMessageLiveLocation`

With specific bot

```
result: Union[Message, bool] = await bot(StopMessageLiveLocation(...))
```

As reply into Webhook in handler

```
return StopMessageLiveLocation(...)
```

As shortcut from received object

- *aiogram.types.message.Message.stop_live_location()*

stopPoll

Returns: `Poll`

class aiogram.methods.stop_poll.**StopPoll**(**, chat_id: int | str, message_id: int, business_connection_id: str | None = None, reply_markup: InlineKeyboardMarkup | None = None, **extra_data: Any*)

Use this method to stop a poll which was sent by the bot. On success, the stopped *aiogram.types.poll.Poll* is returned.

Source: https://core.telegram.org/bots/api#stoppoll

chat_id: int | str

Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

message_id: int

Identifier of the original message with the poll

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message to be edited was sent

reply_markup: *InlineKeyboardMarkup* | None

A JSON-serialized object for a new message inline keyboard.

Usage

As bot method

```
result: Poll = await bot.stop_poll(...)
```

Method as object

Imports:

- `from aiogram.methods.stop_poll import StopPoll`
- alias: `from aiogram.methods import StopPoll`

With specific bot

```
result: Poll = await bot(StopPoll(...))
```

As reply into Webhook in handler

```
return StopPoll(...)
```

Inline mode

answerInlineQuery

Returns: `bool`

class aiogram.methods.answer_inline_query.**AnswerInlineQuery**(*, *inline_query_id: str*, *results: List[*InlineQueryResultCachedAudio | InlineQueryResultCachedDocument | InlineQueryResultCachedGif | InlineQueryResultCachedMpeg4Gif | InlineQueryResultCachedPhoto | InlineQueryResultCachedSticker | InlineQueryResultCachedVideo | InlineQueryResultCachedVoice | InlineQueryResultArticle | InlineQueryResultAudio | InlineQueryResultContact | InlineQueryResultGame | InlineQueryResultDocument | InlineQueryResultGif | InlineQueryResultLocation | InlineQueryResultMpeg4Gif | InlineQueryResultPhoto | InlineQueryResultVenue | InlineQueryResultVideo | InlineQueryResultVoice*]*, *cache_time: int* | *None = None*, *is_personal: bool* | *None = None*, *next_offset: str* | *None = None*, *button:* InlineQueryResultsButton | *None = None*, *switch_pm_parameter: str* | *None = None*, *switch_pm_text: str* | *None = None*, ***extra_data: Any*)

Use this method to send answers to an inline query. On success, `True` is returned.

No more than **50** results per query are allowed.

Source: https://core.telegram.org/bots/api#answerinlinequery

inline_query_id: str

Unique identifier for the answered query

results: List[*InlineQueryResultCachedAudio* | *InlineQueryResultCachedDocument* | *InlineQueryResultCachedGif* | *InlineQueryResultCachedMpeg4Gif* | *InlineQueryResultCachedPhoto* | *InlineQueryResultCachedSticker* | *InlineQueryResultCachedVideo* | *InlineQueryResultCachedVoice* | *InlineQueryResultArticle* | *InlineQueryResultAudio* | *InlineQueryResultContact* | *InlineQueryResultGame* | *InlineQueryResultDocument* | *InlineQueryResultGif* | *InlineQueryResultLocation* | *InlineQueryResultMpeg4Gif* | *InlineQueryResultPhoto* | *InlineQueryResultVenue* | *InlineQueryResultVideo* | *InlineQueryResultVoice*]

A JSON-serialized array of results for the inline query

cache_time: int | None

The maximum amount of time in seconds that the result of the inline query may be cached on the server. Defaults to 300.

is_personal: bool | None

Pass `True` if results may be cached on the server side only for the user that sent the query. By default, results may be returned to any user who sends the same query.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

next_offset: str | None

Pass the offset that a client should send in the next query with the same text to receive more results. Pass an empty string if there are no more results or if you don't support pagination. Offset length can't exceed 64 bytes.

button: *InlineQueryResultsButton* | None

A JSON-serialized object describing a button to be shown above inline query results

switch_pm_parameter: str | None

Deep-linking parameter for the /start message sent to the bot when user presses the switch button. 1-64 characters, only `A-Z`, `a-z`, `0-9`, `_` and `-` are allowed.

Deprecated since version API:6.7: https://core.telegram.org/bots/api-changelog#april-21-2023

switch_pm_text: str | None

If passed, clients will display a button with specified text that switches the user to a private chat with the bot and sends the bot a start message with the parameter *switch_pm_parameter*

Deprecated since version API:6.7: https://core.telegram.org/bots/api-changelog#april-21-2023

Usage

As bot method

```
result: bool = await bot.answer_inline_query(...)
```

Method as object

Imports:

- `from aiogram.methods.answer_inline_query import AnswerInlineQuery`
- alias: `from aiogram.methods import AnswerInlineQuery`

With specific bot

```
result: bool = await bot(AnswerInlineQuery(...))
```

As reply into Webhook in handler

```
return AnswerInlineQuery(...)
```

As shortcut from received object

- *aiogram.types.inline_query.InlineQuery.answer()*

answerWebAppQuery

Returns: `SentWebAppMessage`

class aiogram.methods.answer_web_app_query.**AnswerWebAppQuery**(**, web_app_query_id: str, result: InlineQueryResultCachedAudio | InlineQueryResultCachedDocument | InlineQueryResultCachedGif | InlineQueryResultCachedMpeg4Gif | InlineQueryResultCachedPhoto | InlineQueryResultCachedSticker | InlineQueryResultCachedVideo | InlineQueryResultCachedVoice | InlineQueryResultArticle | InlineQueryResultAudio | InlineQueryResultContact | InlineQueryResultGame | InlineQueryResultDocument | InlineQueryResultGif | InlineQueryResultLocation | InlineQueryResultMpeg4Gif | InlineQueryResultPhoto | InlineQueryResultVenue | InlineQueryResultVideo | InlineQueryResultVoice, **extra_data: Any*)

Use this method to set the result of an interaction with a Web App and send a corresponding message on behalf of the user to the chat from which the query originated. On success, a *aiogram.types.sent_web_app_message.SentWebAppMessage* object is returned.

Source: https://core.telegram.org/bots/api#answerwebappquery

web_app_query_id: str

Unique identifier for the query to be answered

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

result: *InlineQueryResultCachedAudio* | *InlineQueryResultCachedDocument* | *InlineQueryResultCachedGif* | *InlineQueryResultCachedMpeg4Gif* | *InlineQueryResultCachedPhoto* | *InlineQueryResultCachedSticker* | *InlineQueryResultCachedVideo* | *InlineQueryResultCachedVoice* | *InlineQueryResultArticle* | *InlineQueryResultAudio* | *InlineQueryResultContact* | *InlineQueryResultGame* | *InlineQueryResultDocument* | *InlineQueryResultGif* | *InlineQueryResultLocation* | *InlineQueryResultMpeg4Gif* | *InlineQueryResultPhoto* | *InlineQueryResultVenue* | *InlineQueryResultVideo* | *InlineQueryResultVoice*

A JSON-serialized object describing the message to be sent

Usage

As bot method

```
result: SentWebAppMessage = await bot.answer_web_app_query(...)
```

Method as object

Imports:

- `from aiogram.methods.answer_web_app_query import AnswerWebAppQuery`
- alias: `from aiogram.methods import AnswerWebAppQuery`

With specific bot

```
result: SentWebAppMessage = await bot(AnswerWebAppQuery(...))
```

As reply into Webhook in handler

```
return AnswerWebAppQuery(...)
```

Games

getGameHighScores

Returns: `List[GameHighScore]`

class aiogram.methods.get_game_high_scores.**GetGameHighScores**(**, user_id: int, chat_id: int | None = None, message_id: int | None = None, inline_message_id: str | None = None, **extra_data: Any*)

Use this method to get data for high score tables. Will return the score of the specified user and several of their neighbors in a game. Returns an Array of *aiogram.types.game_high_score.GameHighScore* objects.

This method will currently return scores for the target user, plus two of their closest neighbors on each side. Will also return the top three users if the user and their neighbors are not among them. Please note that this behavior is subject to change.

Source: https://core.telegram.org/bots/api#getgamehighscores

user_id: int

Target user id

chat_id: int | None

Required if *inline_message_id* is not specified. Unique identifier for the target chat

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_id: int | None

Required if *inline_message_id* is not specified. Identifier of the sent message

inline_message_id: str | None

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

Usage

As bot method

```
result: List[GameHighScore] = await bot.get_game_high_scores(...)
```

Method as object

Imports:

- `from aiogram.methods.get_game_high_scores import GetGameHighScores`
- alias: `from aiogram.methods import GetGameHighScores`

With specific bot

```
result: List[GameHighScore] = await bot(GetGameHighScores(...))
```

sendGame

Returns: `Message`

class aiogram.methods.send_game.**SendGame**(***, *chat_id: int*, *game_short_name: str*, *business_connection_id: str | None = None*, *message_thread_id: int | None = None*, *disable_notification: bool | None = None*, *protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>*, *message_effect_id: str | None = None*, *reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None*, *reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | None = None*, *allow_sending_without_reply: bool | None = None*, *reply_to_message_id: int | None = None*, ***extra_data: ~typing.Any*)

Use this method to send a game. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendgame

chat_id: int

Unique identifier for the target chat

game_short_name: str

Short name of the game, serves as the unique identifier for the game. Set up your games via @BotFather.

business_connection_id: str | None

Unique identifier of the business connection on behalf of which the message will be sent

message_thread_id: int | None

Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

disable_notification: bool | None

Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None

Protects the contents of the sent message from forwarding and saving

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

message_effect_id: str | None

Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None

Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | None

A JSON-serialized object for an inline keyboard. If empty, one 'Play game_title' button will be shown. If not empty, the first button must launch the game.

allow_sending_without_reply: bool | None

Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None

If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_game(...)
```

Method as object

Imports:

- `from aiogram.methods.send_game import SendGame`
- alias: `from aiogram.methods import SendGame`

With specific bot

```
result: Message = await bot(SendGame(...))
```

As reply into Webhook in handler

```
return SendGame(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_game()*
- *aiogram.types.message.Message.reply_game()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_game()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_game_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_game()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_game()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_game()*

setGameScore

Returns: `Union[Message, bool]`

class `aiogram.methods.set_game_score.`**`SetGameScore`**(**, user_id: int, score: int, force: bool | None = None, disable_edit_message: bool | None = None, chat_id: int | None = None, message_id: int | None = None, inline_message_id: str | None = None, **extra_data: Any*)

Use this method to set the score of the specified user in a game message. On success, if the message is not an inline message, the `aiogram.types.message.Message` is returned, otherwise `True` is returned. Returns an error, if the new score is not greater than the user's current score in the chat and *force* is `False`.

Source: https://core.telegram.org/bots/api#setgamescore

`user_id: int`

User identifier

`score: int`

New score, must be non-negative

`force: bool | None`

Pass `True` if the high score is allowed to decrease. This can be useful when fixing mistakes or banning cheaters

`disable_edit_message: bool | None`

Pass `True` if the game message should not be automatically edited to include the current scoreboard

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`chat_id: int | None`

Required if *inline_message_id* is not specified. Unique identifier for the target chat

`message_id: int | None`

Required if *inline_message_id* is not specified. Identifier of the sent message

`inline_message_id: str | None`

Required if *chat_id* and *message_id* are not specified. Identifier of the inline message

Usage

As bot method

```
result: Union[Message, bool] = await bot.set_game_score(...)
```

Method as object

Imports:

- `from aiogram.methods.set_game_score import SetGameScore`
- alias: `from aiogram.methods import SetGameScore`

With specific bot

```
result: Union[Message, bool] = await bot(SetGameScore(...))
```

As reply into Webhook in handler

```
return SetGameScore(...)
```

Payments

answerPreCheckoutQuery

Returns: `bool`

class `aiogram.methods.answer_pre_checkout_query.`**`AnswerPreCheckoutQuery`**(*, *pre_checkout_query_id: str*, *ok: bool*, *error_message: str | None = None*, ***extra_data: Any*)

Once the user has confirmed their payment and shipping details, the Bot API sends the final confirmation in the form of an `aiogram.types.update.Update` with the field *pre_checkout_query*. Use this method to respond to such pre-checkout queries. On success, `True` is returned. **Note:** The Bot API must receive an answer within 10 seconds after the pre-checkout query was sent.

Source: https://core.telegram.org/bots/api#answerprecheckoutquery

pre_checkout_query_id: str

Unique identifier for the query to be answered

ok: bool

Specify `True` if everything is alright (goods are available, etc.) and the bot is ready to proceed with the order. Use `False` if there are any problems.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

error_message: str | None

Required if *ok* is `False`. Error message in human readable form that explains the reason for failure to proceed with the checkout (e.g. “Sorry, somebody just bought the last of our amazing black T-shirts while you were busy filling out your payment details. Please choose a different color or garment!”). Telegram will display this message to the user.

Usage

As bot method

```
result: bool = await bot.answer_pre_checkout_query(...)
```

Method as object

Imports:

- `from aiogram.methods.answer_pre_checkout_query import AnswerPreCheckoutQuery`
- alias: `from aiogram.methods import AnswerPreCheckoutQuery`

With specific bot

```
result: bool = await bot(AnswerPreCheckoutQuery(...))
```

As reply into Webhook in handler

```
return AnswerPreCheckoutQuery(...)
```

As shortcut from received object

- *`aiogram.types.pre_checkout_query.PreCheckoutQuery.answer()`*

answerShippingQuery

Returns: `bool`

class `aiogram.methods.answer_shipping_query.`**`AnswerShippingQuery`**(**, shipping_query_id: str, ok: bool, shipping_options: List[ShippingOption] | None = None, error_message: str | None = None, **extra_data: Any*)

If you sent an invoice requesting a shipping address and the parameter *is_flexible* was specified, the Bot API will send an *`aiogram.types.update.Update`* with a *shipping_query* field to the bot. Use this method to reply to shipping queries. On success, `True` is returned.

Source: https://core.telegram.org/bots/api#answershippingquery

`shipping_query_id: str`

Unique identifier for the query to be answered

`ok: bool`

Pass `True` if delivery to the specified address is possible and `False` if there are any problems (for example, if delivery to the specified address is not possible)

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

shipping_options: List[*ShippingOption*] | None

Required if *ok* is `True`. A JSON-serialized array of available shipping options.

error_message: str | None

Required if *ok* is `False`. Error message in human readable form that explains why it is impossible to complete the order (e.g. "Sorry, delivery to your desired address is unavailable'). Telegram will display this message to the user.

Usage

As bot method

```
result: bool = await bot.answer_shipping_query(...)
```

Method as object

Imports:

- `from aiogram.methods.answer_shipping_query import AnswerShippingQuery`
- alias: `from aiogram.methods import AnswerShippingQuery`

With specific bot

```
result: bool = await bot(AnswerShippingQuery(...))
```

As reply into Webhook in handler

```
return AnswerShippingQuery(...)
```

As shortcut from received object

- *aiogram.types.shipping_query.ShippingQuery.answer()*

createInvoiceLink

Returns: `str`

class `aiogram.methods.create_invoice_link.`**`CreateInvoiceLink`**(**, title: str, description: str, payload: str, currency: str, prices: List[LabeledPrice], provider_token: str | None = None, max_tip_amount: int | None = None, suggested_tip_amounts: List[int] | None = None, provider_data: str | None = None, photo_url: str | None = None, photo_size: int | None = None, photo_width: int | None = None, photo_height: int | None = None, need_name: bool | None = None, need_phone_number: bool | None = None, need_email: bool | None = None, need_shipping_address: bool | None = None, send_phone_number_to_provider: bool | None = None, send_email_to_provider: bool | None = None, is_flexible: bool | None = None, **extra_data: Any*)

Use this method to create a link for an invoice. Returns the created invoice link as *String* on success.

Source: https://core.telegram.org/bots/api#createinvoicelink

`title: str`

Product name, 1-32 characters

`description: str`

Product description, 1-255 characters

`payload: str`

Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.

`currency: str`

Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.

`prices: List[`*`LabeledPrice`*`]`

Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.

`provider_token: str | None`

Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.

`max_tip_amount: int | None`

The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.

suggested_tip_amounts: List[int] | None

A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.

provider_data: str | None

JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.

photo_url: str | None

URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

photo_size: int | None

Photo size in bytes

photo_width: int | None

Photo width

photo_height: int | None

Photo height

need_name: bool | None

Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.

need_phone_number: bool | None

Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.

need_email: bool | None

Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.

need_shipping_address: bool | None

Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.

send_phone_number_to_provider: bool | None

Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.

send_email_to_provider: bool | None

Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.

is_flexible: bool | None

Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.

Usage

As bot method

```
result: str = await bot.create_invoice_link(...)
```

Method as object

Imports:

- `from aiogram.methods.create_invoice_link import CreateInvoiceLink`
- alias: `from aiogram.methods import CreateInvoiceLink`

With specific bot

```
result: str = await bot(CreateInvoiceLink(...))
```

As reply into Webhook in handler

```
return CreateInvoiceLink(...)
```

getStarTransactions

Returns: `StarTransactions`

class `aiogram.methods.get_star_transactions.`**`GetStarTransactions`**`(*, offset: int | None = None, limit: int | None = None, **extra_data: Any)`

Returns the bot's Telegram Star transactions in chronological order. On success, returns a *`aiogram.types.star_transactions.StarTransactions`* object.

Source: https://core.telegram.org/bots/api#getstartransactions

`offset: int | None`

Number of transactions to skip in the response

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init``(context: Any, /)` → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`limit: int | None`

The maximum number of transactions to be retrieved. Values between 1-100 are accepted. Defaults to 100.

Usage

As bot method

```
result: StarTransactions = await bot.get_star_transactions(...)
```

Method as object

Imports:

- `from aiogram.methods.get_star_transactions import GetStarTransactions`
- alias: `from aiogram.methods import GetStarTransactions`

With specific bot

```
result: StarTransactions = await bot(GetStarTransactions(...))
```

refundStarPayment

Returns: `bool`

class `aiogram.methods.refund_star_payment.`**`RefundStarPayment`**(*, *user_id: int*, *telegram_payment_charge_id: str*, ***extra_data: Any*)

Refunds a successful payment in Telegram Stars. Returns `True` on success.

Source: https://core.telegram.org/bots/api#refundstarpayment

`user_id: int`

Identifier of the user whose payment will be refunded

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`telegram_payment_charge_id: str`

Telegram payment identifier

Usage

As bot method

```
result: bool = await bot.refund_star_payment(...)
```

Method as object

Imports:

- `from aiogram.methods.refund_star_payment import RefundStarPayment`
- alias: `from aiogram.methods import RefundStarPayment`

With specific bot

```
result: bool = await bot(RefundStarPayment(...))
```

As reply into Webhook in handler

```
return RefundStarPayment(...)
```

sendInvoice

Returns: `Message`

class `aiogram.methods.send_invoice.`**`SendInvoice`**(**, chat_id: int | str, title: str, description: str, payload: str, currency: str, prices: ~typing.List[~aiogram.types.labeled_price.LabeledPrice], message_thread_id: int | None = None, provider_token: str | None = None, max_tip_amount: int | None = None, suggested_tip_amounts: ~typing.List[int] | None = None, start_parameter: str | None = None, provider_data: str | None = None, photo_url: str | None = None, photo_size: int | None = None, photo_width: int | None = None, photo_height: int | None = None, need_name: bool | None = None, need_phone_number: bool | None = None, need_email: bool | None = None, need_shipping_address: bool | None = None, send_phone_number_to_provider: bool | None = None, send_email_to_provider: bool | None = None, is_flexible: bool | None = None, disable_notification: bool | None = None, protect_content: bool | ~aiogram.client.default.Default | None = <Default('protect_content')>, message_effect_id: str | None = None, reply_parameters: ~aiogram.types.reply_parameters.ReplyParameters | None = None, reply_markup: ~aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup | None = None, allow_sending_without_reply: bool | None = None, reply_to_message_id: int | None = None, **extra_data: ~typing.Any*)

Use this method to send invoices. On success, the sent `aiogram.types.message.Message` is returned.

Source: https://core.telegram.org/bots/api#sendinvoice

chat_id: int | str
: Unique identifier for the target chat or username of the target channel (in the format `@channelusername`)

title: str
: Product name, 1-32 characters

description: str
: Product description, 1-255 characters

payload: str
: Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use it for your internal processes.

currency: str
: Three-letter ISO 4217 currency code, see more on currencies. Pass 'XTR' for payments in Telegram Stars.

prices: List[*LabeledPrice*]
: Price breakdown, a JSON-serialized list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.). Must contain exactly one item for payments in Telegram Stars.

message_thread_id: int | None
: Unique identifier for the target message thread (topic) of the forum; for forum supergroups only

provider_token: str | None
: Payment provider token, obtained via @BotFather. Pass an empty string for payments in Telegram Stars.

max_tip_amount: int | None
: The maximum accepted amount for tips in the *smallest units* of the currency (integer, **not** float/double). For example, for a maximum tip of `US$ 1.45` pass `max_tip_amount = 145`. See the *exp* parameter in currencies.json, it shows the number of digits past the decimal point for each currency (2 for the majority of currencies). Defaults to 0. Not supported for payments in Telegram Stars.

suggested_tip_amounts: List[int] | None
: A JSON-serialized array of suggested amounts of tips in the *smallest units* of the currency (integer, **not** float/double). At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed *max_tip_amount*.

start_parameter: str | None
: Unique deep-linking parameter. If left empty, **forwarded copies** of the sent message will have a *Pay* button, allowing multiple users to pay directly from the forwarded message, using the same invoice. If non-empty, forwarded copies of the sent message will have a *URL* button with a deep link to the bot (instead of a *Pay* button), with the value used as the start parameter

provider_data: str | None
: JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.

photo_url: str | None
: URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.

photo_size: int | None
: Photo size in bytes

photo_width: int | None
: Photo width

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
We need to both initialize private attributes and call the user-defined model_post_init method.

photo_height: int | None
Photo height

need_name: bool | None
Pass `True` if you require the user's full name to complete the order. Ignored for payments in Telegram Stars.

need_phone_number: bool | None
Pass `True` if you require the user's phone number to complete the order. Ignored for payments in Telegram Stars.

need_email: bool | None
Pass `True` if you require the user's email address to complete the order. Ignored for payments in Telegram Stars.

need_shipping_address: bool | None
Pass `True` if you require the user's shipping address to complete the order. Ignored for payments in Telegram Stars.

send_phone_number_to_provider: bool | None
Pass `True` if the user's phone number should be sent to the provider. Ignored for payments in Telegram Stars.

send_email_to_provider: bool | None
Pass `True` if the user's email address should be sent to the provider. Ignored for payments in Telegram Stars.

is_flexible: bool | None
Pass `True` if the final price depends on the shipping method. Ignored for payments in Telegram Stars.

disable_notification: bool | None
Sends the message silently. Users will receive a notification with no sound.

protect_content: bool | Default | None
Protects the contents of the sent message from forwarding and saving

message_effect_id: str | None
Unique identifier of the message effect to be added to the message; for private chats only

reply_parameters: *ReplyParameters* | None
Description of the message to reply to

reply_markup: *InlineKeyboardMarkup* | None
A JSON-serialized object for an inline keyboard. If empty, one 'Pay `total price`' button will be shown. If not empty, the first button must be a Pay button.

allow_sending_without_reply: bool | None
Pass `True` if the message should be sent even if the specified replied-to message is not found

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

reply_to_message_id: int | None
If the message is a reply, ID of the original message

Deprecated since version API:7.0: https://core.telegram.org/bots/api-changelog#december-29-2023

Usage

As bot method

```
result: Message = await bot.send_invoice(...)
```

Method as object

Imports:

- `from aiogram.methods.send_invoice import SendInvoice`
- alias: `from aiogram.methods import SendInvoice`

With specific bot

```
result: Message = await bot(SendInvoice(...))
```

As reply into Webhook in handler

```
return SendInvoice(...)
```

As shortcut from received object

- *aiogram.types.message.Message.answer_invoice()*
- *aiogram.types.message.Message.reply_invoice()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_invoice()*
- *aiogram.types.chat_join_request.ChatJoinRequest.answer_invoice_pm()*
- *aiogram.types.chat_member_updated.ChatMemberUpdated.answer_invoice()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.answer_invoice()*
- *aiogram.types.inaccessible_message.InaccessibleMessage.reply_invoice()*

Getting updates

deleteWebhook

Returns: `bool`

class `aiogram.methods.delete_webhook.`**`DeleteWebhook`**`(*, drop_pending_updates: bool | None = None, **extra_data: Any)`

Use this method to remove webhook integration if you decide to switch back to *aiogram.methods.get_updates.GetUpdates*. Returns `True` on success.

Source: https://core.telegram.org/bots/api#deletewebhook

drop_pending_updates: bool | None
: Pass `True` to drop all pending updates

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}
: A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None
: We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: bool = await bot.delete_webhook(...)
```

Method as object

Imports:

- `from aiogram.methods.delete_webhook import DeleteWebhook`
- alias: `from aiogram.methods import DeleteWebhook`

With specific bot

```
result: bool = await bot(DeleteWebhook(...))
```

As reply into Webhook in handler

```
return DeleteWebhook(...)
```

getUpdates

Returns: `List[Update]`

class `aiogram.methods.get_updates.`**GetUpdates**(**, offset: int | None = None, limit: int | None = None, timeout: int | None = None, allowed_updates: List[str] | None = None, **extra_data: Any*)

Use this method to receive incoming updates using long polling (wiki). Returns an Array of *`aiogram.types.update.Update`* objects.

Notes

1. This method will not work if an outgoing webhook is set up.

2. In order to avoid getting duplicate updates, recalculate *offset* after each server response.

Source: https://core.telegram.org/bots/api#getupdates

offset: int | None

Identifier of the first update to be returned. Must be greater by one than the highest among the identifiers of previously received updates. By default, updates starting with the earliest unconfirmed update are returned. An update is considered confirmed as soon as `aiogram.methods.get_updates.GetUpdates` is called with an *offset* higher than its *update_id*. The negative offset can be specified to retrieve updates starting from *-offset* update from the end of the updates queue. All previous updates will be forgotten.

limit: int | None

Limits the number of updates to be retrieved. Values between 1-100 are accepted. Defaults to 100.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

timeout: int | None

Timeout in seconds for long polling. Defaults to 0, i.e. usual short polling. Should be positive, short polling should be used for testing purposes only.

allowed_updates: List[str] | None

A JSON-serialized list of the update types you want your bot to receive. For example, specify `["message", "edited_channel_post", "callback_query"]` to only receive updates of these types. See `aiogram.types.update.Update` for a complete list of available update types. Specify an empty list to receive all update types except *chat_member*, *message_reaction*, and *message_reaction_count* (default). If not specified, the previous setting will be used.

Usage

As bot method

```
result: List[Update] = await bot.get_updates(...)
```

Method as object

Imports:

- `from aiogram.methods.get_updates import GetUpdates`
- alias: `from aiogram.methods import GetUpdates`

With specific bot

```
result: List[Update] = await bot(GetUpdates(...))
```

getWebhookInfo

Returns: `WebhookInfo`

class `aiogram.methods.get_webhook_info.`**`GetWebhookInfo`**(***extra_data: Any*)

Use this method to get current webhook status. Requires no parameters. On success, returns a `aiogram.types.webhook_info.WebhookInfo` object. If the bot is using `aiogram.methods.get_updates.GetUpdates`, will return an object with the *url* field empty.

Source: https://core.telegram.org/bots/api#getwebhookinfo

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

Usage

As bot method

```
result: WebhookInfo = await bot.get_webhook_info(...)
```

Method as object

Imports:

- `from aiogram.methods.get_webhook_info import GetWebhookInfo`
- alias: `from aiogram.methods import GetWebhookInfo`

With specific bot

```
result: WebhookInfo = await bot(GetWebhookInfo(...))
```

setWebhook

Returns: `bool`

class `aiogram.methods.set_webhook.`**`SetWebhook`**(***, *url: str*, *certificate: InputFile | None = None*, *ip_address: str | None = None*, *max_connections: int | None = None*, *allowed_updates: List[str] | None = None*, *drop_pending_updates: bool | None = None*, *secret_token: str | None = None*, ***extra_data: Any*)

Use this method to specify a URL and receive incoming updates via an outgoing webhook. Whenever there is an update for the bot, we will send an HTTPS POST request to the specified URL, containing a JSON-serialized `aiogram.types.update.Update`. In case of an unsuccessful request, we will give up after a reasonable amount of attempts. Returns `True` on success. If you'd like to make sure that the webhook was set by you, you can specify secret data in the parameter *secret_token*. If specified, the request will contain a header 'X-Telegram-Bot-Api-Secret-Token' with the secret token as content.

Notes

1. You will not be able to receive updates using `aiogram.methods.get_updates.GetUpdates` for as long as an outgoing webhook is set up.

2. To use a self-signed certificate, you need to upload your public key certificate using *certificate* parameter. Please upload as InputFile, sending a String will not work.

3. Ports currently supported *for webhooks*: **443, 80, 88, 8443**. If you're having any trouble setting up webhooks, please check out this amazing guide to webhooks.

Source: https://core.telegram.org/bots/api#setwebhook

`url: str`

HTTPS URL to send updates to. Use an empty string to remove webhook integration

`certificate: InputFile | None`

Upload your public key certificate so that the root certificate in use can be checked. See our self-signed guide for details.

`ip_address: str | None`

The fixed IP address which will be used to send webhook requests instead of the IP address resolved through DNS

`max_connections: int | None`

The maximum allowed number of simultaneous HTTPS connections to the webhook for update delivery, 1-100. Defaults to *40*. Use lower values to limit the load on your bot's server, and higher values to increase your bot's throughput.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`allowed_updates: List[str] | None`

A JSON-serialized list of the update types you want your bot to receive. For example, specify `["message", "edited_channel_post", "callback_query"]` to only receive updates of these types. See `aiogram.types.update.Update` for a complete list of available update types. Specify an empty list to receive all update types except *chat_member*, *message_reaction*, and *message_reaction_count* (default). If not specified, the previous setting will be used.

`drop_pending_updates: bool | None`

Pass `True` to drop all pending updates

`secret_token: str | None`

A secret token to be sent in a header 'X-Telegram-Bot-Api-Secret-Token' in every webhook request, 1-256 characters. Only characters `A-Z`, `a-z`, `0-9`, `_` and `-` are allowed. The header is useful to ensure that the request comes from a webhook set by you.

Usage

As bot method

```
result: bool = await bot.set_webhook(...)
```

Method as object

Imports:

- `from aiogram.methods.set_webhook import SetWebhook`
- alias: `from aiogram.methods import SetWebhook`

With specific bot

```
result: bool = await bot(SetWebhook(...))
```

As reply into Webhook in handler

```
return SetWebhook(...)
```

Telegram Passport

setPassportDataErrors

Returns: `bool`

class `aiogram.methods.set_passport_data_errors.`**SetPassportDataErrors**(*, *user_id: int*, *errors: List[*PassportElementErrorDataField | PassportElementErrorFrontSide | PassportElementErrorReverseSide | PassportElementErrorSelfie | PassportElementErrorFile | PassportElementErrorFiles | PassportElementErrorTranslationFile | PassportElementErrorTranslationFiles | PassportElementErrorUnspecified*]*, ***extra_data: Any*)

Informs a user that some of the Telegram Passport elements they provided contains errors. The user will not be able to re-submit their Passport to you until the errors are fixed (the contents of the field for which you returned the error must change). Returns `True` on success. Use this if the data submitted by the user doesn't satisfy the standards your service requires for any reason. For example, if a birthday date seems invalid, a submitted document is blurry, a scan shows evidence of tampering, etc. Supply some details in the error message to make sure the user knows how to correct the issues.

Source: https://core.telegram.org/bots/api#setpassportdataerrors

user_id: int

User identifier

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

errors: List[*PassportElementErrorDataField* | *PassportElementErrorFrontSide* | *PassportElementErrorReverseSide* | *PassportElementErrorSelfie* | *PassportElementErrorFile* | *PassportElementErrorFiles* | *PassportElementErrorTranslationFile* | *PassportElementErrorTranslationFiles* | *PassportElementErrorUnspecified*]

A JSON-serialized array describing the errors

Usage

As bot method

```
result: bool = await bot.set_passport_data_errors(...)
```

Method as object

Imports:

- `from aiogram.methods.set_passport_data_errors import SetPassportDataErrors`
- alias: `from aiogram.methods import SetPassportDataErrors`

With specific bot

```
result: bool = await bot(SetPassportDataErrors(...))
```

As reply into Webhook in handler

```
return SetPassportDataErrors(...)
```

2.3.5 Enums

Here is list of all available enums:

BotCommandScopeType

class aiogram.enums.bot_command_scope_type.**BotCommandScopeType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents the scope to which bot commands are applied.

Source: https://core.telegram.org/bots/api#botcommandscope

DEFAULT = 'default'

ALL_PRIVATE_CHATS = 'all_private_chats'

ALL_GROUP_CHATS = 'all_group_chats'

ALL_CHAT_ADMINISTRATORS = 'all_chat_administrators'

CHAT = 'chat'

CHAT_ADMINISTRATORS = 'chat_administrators'

CHAT_MEMBER = 'chat_member'

ChatAction

class aiogram.enums.chat_action.**ChatAction**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents bot actions.

Choose one, depending on what the user is about to receive:

- typing for text messages,
- upload_photo for photos,
- record_video or upload_video for videos,
- record_voice or upload_voice for voice notes,
- upload_document for general files,
- choose_sticker for stickers,
- find_location for location data,
- record_video_note or upload_video_note for video notes.

Source: https://core.telegram.org/bots/api#sendchataction

TYPING = 'typing'

UPLOAD_PHOTO = 'upload_photo'

RECORD_VIDEO = 'record_video'

`UPLOAD_VIDEO = 'upload_video'`

`RECORD_VOICE = 'record_voice'`

`UPLOAD_VOICE = 'upload_voice'`

`UPLOAD_DOCUMENT = 'upload_document'`

`CHOOSE_STICKER = 'choose_sticker'`

`FIND_LOCATION = 'find_location'`

`RECORD_VIDEO_NOTE = 'record_video_note'`

`UPLOAD_VIDEO_NOTE = 'upload_video_note'`

ChatBoostSourceType

class `aiogram.enums.chat_boost_source_type.`**`ChatBoostSourceType`**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents a type of chat boost source.

Source: https://core.telegram.org/bots/api#chatboostsource

`PREMIUM = 'premium'`

`GIFT_CODE = 'gift_code'`

`GIVEAWAY = 'giveaway'`

ChatMemberStatus

class `aiogram.enums.chat_member_status.`**`ChatMemberStatus`**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents chat member status.

Source: https://core.telegram.org/bots/api#chatmember

`CREATOR = 'creator'`

`ADMINISTRATOR = 'administrator'`

`MEMBER = 'member'`

`RESTRICTED = 'restricted'`

`LEFT = 'left'`

`KICKED = 'kicked'`

ChatType

class aiogram.enums.chat_type.**ChatType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents a chat type

Source: https://core.telegram.org/bots/api#chat

SENDER = 'sender'

PRIVATE = 'private'

GROUP = 'group'

SUPERGROUP = 'supergroup'

CHANNEL = 'channel'

ContentType

class aiogram.enums.content_type.**ContentType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents a type of content in message

UNKNOWN = 'unknown'

ANY = 'any'

TEXT = 'text'

ANIMATION = 'animation'

AUDIO = 'audio'

DOCUMENT = 'document'

PAID_MEDIA = 'paid_media'

PHOTO = 'photo'

STICKER = 'sticker'

STORY = 'story'

VIDEO = 'video'

VIDEO_NOTE = 'video_note'

VOICE = 'voice'

CONTACT = 'contact'

DICE = 'dice'

GAME = 'game'

POLL = 'poll'

VENUE = 'venue'

LOCATION = 'location'

NEW_CHAT_MEMBERS = 'new_chat_members'

LEFT_CHAT_MEMBER = 'left_chat_member'

NEW_CHAT_TITLE = 'new_chat_title'

NEW_CHAT_PHOTO = 'new_chat_photo'

DELETE_CHAT_PHOTO = 'delete_chat_photo'

GROUP_CHAT_CREATED = 'group_chat_created'

SUPERGROUP_CHAT_CREATED = 'supergroup_chat_created'

CHANNEL_CHAT_CREATED = 'channel_chat_created'

MESSAGE_AUTO_DELETE_TIMER_CHANGED = 'message_auto_delete_timer_changed'

MIGRATE_TO_CHAT_ID = 'migrate_to_chat_id'

MIGRATE_FROM_CHAT_ID = 'migrate_from_chat_id'

PINNED_MESSAGE = 'pinned_message'

INVOICE = 'invoice'

SUCCESSFUL_PAYMENT = 'successful_payment'

REFUNDED_PAYMENT = 'refunded_payment'

USERS_SHARED = 'users_shared'

CHAT_SHARED = 'chat_shared'

CONNECTED_WEBSITE = 'connected_website'

WRITE_ACCESS_ALLOWED = 'write_access_allowed'

PASSPORT_DATA = 'passport_data'

PROXIMITY_ALERT_TRIGGERED = 'proximity_alert_triggered'

BOOST_ADDED = 'boost_added'

CHAT_BACKGROUND_SET = 'chat_background_set'

FORUM_TOPIC_CREATED = 'forum_topic_created'

FORUM_TOPIC_EDITED = 'forum_topic_edited'

FORUM_TOPIC_CLOSED = 'forum_topic_closed'

FORUM_TOPIC_REOPENED = 'forum_topic_reopened'

GENERAL_FORUM_TOPIC_HIDDEN = 'general_forum_topic_hidden'

GENERAL_FORUM_TOPIC_UNHIDDEN = 'general_forum_topic_unhidden'

`GIVEAWAY_CREATED = 'giveaway_created'`

`GIVEAWAY = 'giveaway'`

`GIVEAWAY_WINNERS = 'giveaway_winners'`

`GIVEAWAY_COMPLETED = 'giveaway_completed'`

`VIDEO_CHAT_SCHEDULED = 'video_chat_scheduled'`

`VIDEO_CHAT_STARTED = 'video_chat_started'`

`VIDEO_CHAT_ENDED = 'video_chat_ended'`

`VIDEO_CHAT_PARTICIPANTS_INVITED = 'video_chat_participants_invited'`

`WEB_APP_DATA = 'web_app_data'`

`USER_SHARED = 'user_shared'`

Currency

`class aiogram.enums.currency.Currency`(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Currencies supported by Telegram Bot API

Source: https://core.telegram.org/bots/payments#supported-currencies

`AED = 'AED'`

`AFN = 'AFN'`

`ALL = 'ALL'`

`AMD = 'AMD'`

`ARS = 'ARS'`

`AUD = 'AUD'`

`AZN = 'AZN'`

`BAM = 'BAM'`

`BDT = 'BDT'`

`BGN = 'BGN'`

`BND = 'BND'`

`BOB = 'BOB'`

`BRL = 'BRL'`

`BYN = 'BYN'`

`CAD = 'CAD'`

`CHF = 'CHF'`

CLP = 'CLP'

CNY = 'CNY'

COP = 'COP'

CRC = 'CRC'

CZK = 'CZK'

DKK = 'DKK'

DOP = 'DOP'

DZD = 'DZD'

EGP = 'EGP'

ETB = 'ETB'

EUR = 'EUR'

GBP = 'GBP'

GEL = 'GEL'

GTQ = 'GTQ'

HKD = 'HKD'

HNL = 'HNL'

HRK = 'HRK'

HUF = 'HUF'

IDR = 'IDR'

ILS = 'ILS'

INR = 'INR'

ISK = 'ISK'

JMD = 'JMD'

JPY = 'JPY'

KES = 'KES'

KGS = 'KGS'

KRW = 'KRW'

KZT = 'KZT'

LBP = 'LBP'

LKR = 'LKR'

MAD = 'MAD'

MDL = 'MDL'

MNT = 'MNT'

MUR = 'MUR'

MVR = 'MVR'

MXN = 'MXN'

MYR = 'MYR'

MZN = 'MZN'

NGN = 'NGN'

NIO = 'NIO'

NOK = 'NOK'

NPR = 'NPR'

NZD = 'NZD'

PAB = 'PAB'

PEN = 'PEN'

PHP = 'PHP'

PKR = 'PKR'

PLN = 'PLN'

PYG = 'PYG'

QAR = 'QAR'

RON = 'RON'

RSD = 'RSD'

RUB = 'RUB'

SAR = 'SAR'

SEK = 'SEK'

SGD = 'SGD'

THB = 'THB'

TJS = 'TJS'

TRY = 'TRY'

TTD = 'TTD'

TWD = 'TWD'

TZS = 'TZS'

`UAH = 'UAH'`

`UGX = 'UGX'`

`USD = 'USD'`

`UYU = 'UYU'`

`UZS = 'UZS'`

`VND = 'VND'`

`YER = 'YER'`

`ZAR = 'ZAR'`

DiceEmoji

class `aiogram.enums.dice_emoji.`**`DiceEmoji`**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Emoji on which the dice throw animation is based

Source: https://core.telegram.org/bots/api#dice

`DICE = ''`

`DART = ''`

`BASKETBALL = ''`

`FOOTBALL = ''`

`SLOT_MACHINE = ''`

`BOWLING = ''`

EncryptedPassportElement

class `aiogram.enums.encrypted_passport_element.`**`EncryptedPassportElement`**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents type of encrypted passport element.

Source: https://core.telegram.org/bots/api#encryptedpassportelement

`PERSONAL_DETAILS = 'personal_details'`

`PASSPORT = 'passport'`

`DRIVER_LICENSE = 'driver_license'`

`IDENTITY_CARD = 'identity_card'`

`INTERNAL_PASSPORT = 'internal_passport'`

`ADDRESS = 'address'`

`UTILITY_BILL = 'utility_bill'`

`BANK_STATEMENT = 'bank_statement'`

`RENTAL_AGREEMENT = 'rental_agreement'`

`PASSPORT_REGISTRATION = 'passport_registration'`

`TEMPORARY_REGISTRATION = 'temporary_registration'`

`PHONE_NUMBER = 'phone_number'`

`EMAIL = 'email'`

InlineQueryResultType

class `aiogram.enums.inline_query_result_type.`**`InlineQueryResultType`**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Type of inline query result

Source: https://core.telegram.org/bots/api#inlinequeryresult

`AUDIO = 'audio'`

`DOCUMENT = 'document'`

`GIF = 'gif'`

`MPEG4_GIF = 'mpeg4_gif'`

`PHOTO = 'photo'`

`STICKER = 'sticker'`

`VIDEO = 'video'`

`VOICE = 'voice'`

`ARTICLE = 'article'`

`CONTACT = 'contact'`

`GAME = 'game'`

`LOCATION = 'location'`

`VENUE = 'venue'`

InputMediaType

class aiogram.enums.input_media_type.**InputMediaType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents input media type

Source: https://core.telegram.org/bots/api#inputmedia

ANIMATION = 'animation'

AUDIO = 'audio'

DOCUMENT = 'document'

PHOTO = 'photo'

VIDEO = 'video'

InputPaidMediaType

class aiogram.enums.input_paid_media_type.**InputPaidMediaType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents the type of a media in a paid message.

Source: https://core.telegram.org/bots/api#inputpaidmedia

PHOTO = 'photo'

VIDEO = 'video'

KeyboardButtonPollTypeType

class aiogram.enums.keyboard_button_poll_type_type.**KeyboardButtonPollTypeType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents type of a poll, which is allowed to be created and sent when the corresponding button is pressed.

Source: https://core.telegram.org/bots/api#keyboardbuttonpolltype

QUIZ = 'quiz'

REGULAR = 'regular'

MaskPositionPoint

class aiogram.enums.mask_position_point.**MaskPositionPoint**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The part of the face relative to which the mask should be placed.

Source: https://core.telegram.org/bots/api#maskposition

FOREHEAD = 'forehead'

EYES = 'eyes'

MOUTH = 'mouth'

CHIN = 'chin'

MenuButtonType

class aiogram.enums.menu_button_type.**MenuButtonType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents an type of Menu button

Source: https://core.telegram.org/bots/api#menubuttondefault

DEFAULT = 'default'

COMMANDS = 'commands'

WEB_APP = 'web_app'

MessageEntityType

class aiogram.enums.message_entity_type.**MessageEntityType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents type of message entity

Source: https://core.telegram.org/bots/api#messageentity

MENTION = 'mention'

HASHTAG = 'hashtag'

CASHTAG = 'cashtag'

BOT_COMMAND = 'bot_command'

URL = 'url'

EMAIL = 'email'

PHONE_NUMBER = 'phone_number'

BOLD = 'bold'

`ITALIC = 'italic'`

`UNDERLINE = 'underline'`

`STRIKETHROUGH = 'strikethrough'`

`SPOILER = 'spoiler'`

`BLOCKQUOTE = 'blockquote'`

`EXPANDABLE_BLOCKQUOTE = 'expandable_blockquote'`

`CODE = 'code'`

`PRE = 'pre'`

`TEXT_LINK = 'text_link'`

`TEXT_MENTION = 'text_mention'`

`CUSTOM_EMOJI = 'custom_emoji'`

MessageOriginType

class aiogram.enums.message_origin_type.**MessageOriginType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents origin of a message.

Source: https://core.telegram.org/bots/api#messageorigin

`USER = 'user'`

`HIDDEN_USER = 'hidden_user'`

`CHAT = 'chat'`

`CHANNEL = 'channel'`

PaidMediaType

class aiogram.enums.paid_media_type.**PaidMediaType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents the type of a media in a paid message.

Source: https://core.telegram.org/bots/api#paidmedia

`PHOTO = 'photo'`

`PREVIEW = 'preview'`

`VIDEO = 'video'`

ParseMode

`class aiogram.enums.parse_mode.`**`ParseMode`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Formatting options

Source: https://core.telegram.org/bots/api#formatting-options

`MARKDOWN_V2 = 'MarkdownV2'`

`MARKDOWN = 'Markdown'`

`HTML = 'HTML'`

PassportElementErrorType

`class aiogram.enums.passport_element_error_type.`**`PassportElementErrorType`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

This object represents a passport element error type.

Source: https://core.telegram.org/bots/api#passportelementerror

`DATA = 'data'`

`FRONT_SIDE = 'front_side'`

`REVERSE_SIDE = 'reverse_side'`

`SELFIE = 'selfie'`

`FILE = 'file'`

`FILES = 'files'`

`TRANSLATION_FILE = 'translation_file'`

`TRANSLATION_FILES = 'translation_files'`

`UNSPECIFIED = 'unspecified'`

PollType

`class aiogram.enums.poll_type.`**`PollType`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

This object represents poll type

Source: https://core.telegram.org/bots/api#poll

`REGULAR = 'regular'`

`QUIZ = 'quiz'`

ReactionTypeType

class aiogram.enums.reaction_type_type.**ReactionTypeType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents reaction type.

Source: https://core.telegram.org/bots/api#reactiontype

EMOJI = 'emoji'

CUSTOM_EMOJI = 'custom_emoji'

RevenueWithdrawalStateType

class aiogram.enums.revenue_withdrawal_state_type.**RevenueWithdrawalStateType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents a revenue withdrawal state type

Source: https://core.telegram.org/bots/api#revenuewithdrawalstate

FAILED = 'failed'

PENDING = 'pending'

SUCCEEDED = 'succeeded'

StickerFormat

class aiogram.enums.sticker_format.**StickerFormat**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Format of the sticker

Source: https://core.telegram.org/bots/api#createnewstickerset

STATIC = 'static'

ANIMATED = 'animated'

VIDEO = 'video'

StickerType

class aiogram.enums.sticker_type.**StickerType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The part of the face relative to which the mask should be placed.

Source: https://core.telegram.org/bots/api#maskposition

REGULAR = 'regular'

MASK = 'mask'

CUSTOM_EMOJI = 'custom_emoji'

TopicIconColor

class aiogram.enums.topic_icon_color.**TopicIconColor**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Color of the topic icon in RGB format.

Source: https://github.com/telegramdesktop/tdesktop/blob/991fe491c5ae62705d77aa8fdd44a79caf639c45/Telegram/SourceFiles/data/data_forum_topic.cpp#L51-L56

BLUE = 7322096

YELLOW = 16766590

VIOLET = 13338331

GREEN = 9367192

ROSE = 16749490

RED = 16478047

TransactionPartnerType

class aiogram.enums.transaction_partner_type.**TransactionPartnerType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents a type of transaction partner.

Source: https://core.telegram.org/bots/api#transactionpartner

FRAGMENT = 'fragment'

OTHER = 'other'

USER = 'user'

TELEGRAM_ADS = 'telegram_ads'

UpdateType

class aiogram.enums.update_type.**UpdateType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This object represents the complete list of allowed update types

Source: https://core.telegram.org/bots/api#update

MESSAGE = 'message'

EDITED_MESSAGE = 'edited_message'

CHANNEL_POST = 'channel_post'

EDITED_CHANNEL_POST = 'edited_channel_post'

BUSINESS_CONNECTION = 'business_connection'

BUSINESS_MESSAGE = 'business_message'

EDITED_BUSINESS_MESSAGE = 'edited_business_message'

DELETED_BUSINESS_MESSAGES = 'deleted_business_messages'

MESSAGE_REACTION = 'message_reaction'

MESSAGE_REACTION_COUNT = 'message_reaction_count'

INLINE_QUERY = 'inline_query'

CHOSEN_INLINE_RESULT = 'chosen_inline_result'

CALLBACK_QUERY = 'callback_query'

SHIPPING_QUERY = 'shipping_query'

PRE_CHECKOUT_QUERY = 'pre_checkout_query'

PURCHASED_PAID_MEDIA = 'purchased_paid_media'

POLL = 'poll'

POLL_ANSWER = 'poll_answer'

MY_CHAT_MEMBER = 'my_chat_member'

CHAT_MEMBER = 'chat_member'

CHAT_JOIN_REQUEST = 'chat_join_request'

CHAT_BOOST = 'chat_boost'

REMOVED_CHAT_BOOST = 'removed_chat_boost'

2.3.6 How to download file?

Download file manually

First, you must get the *file_id* of the file you want to download. Information about files sent to the bot is contained in Message.

For example, download the document that came to the bot.

```
file_id = message.document.file_id
```

Then use the getFile method to get *file_path*.

```
file = await bot.get_file(file_id)
file_path = file.file_path
```

After that, use the *download_file* method from the bot object.

download_file(...)

Download file by *file_path* to destination.

If you want to automatically create destination (`io.BytesIO`) use default value of destination and handle result of this method.

async Bot.**download_file**(*file_path: str*, *destination: BinaryIO | Path | str | None = None*, *timeout: int = 30*, *chunk_size: int = 65536*, *seek: bool = True*) → BinaryIO | None

Download file by file_path to destination.

If you want to automatically create destination (`io.BytesIO`) use default value of destination and handle result of this method.

Parameters

- **file_path** – File path on Telegram server (You can get it from `aiogram.types.File`)
- **destination** – Filename, file path or instance of `io.IOBase`. For e.g. `io.BytesIO`, defaults to None
- **timeout** – Total timeout in seconds, defaults to 30
- **chunk_size** – File chunks size, defaults to 64 kb
- **seek** – Go to start of file when downloading is finished. Used only for destination with `typing.BinaryIO` type, defaults to True

There are two options where you can download the file: to **disk** or to **binary I/O object**.

Download file to disk

To download file to disk, you must specify the file name or path where to download the file. In this case, the function will return nothing.

```
await bot.download_file(file_path, "text.txt")
```

Download file to binary I/O object

To download file to binary I/O object, you must specify an object with the `typing.BinaryIO` type or use the default (`None`) value.

In the first case, the function will return your object:

```
my_object = MyBinaryIO()
result: MyBinaryIO = await bot.download_file(file_path, my_object)
# print(result is my_object)  # True
```

If you leave the default value, an `io.BytesIO` object will be created and returned.

```
result: io.BytesIO = await bot.download_file(file_path)
```

Download file in short way

Getting *file_path* manually every time is boring, so you should use the *download* method.

download(...)

Download file by *file_id* or *Downloadable* object to destination.

If you want to automatically create destination (`io.BytesIO`) use default value of destination and handle result of this method.

async Bot.download(*file: str | Downloadable*, *destination: BinaryIO | Path | str | None = None*, *timeout: int = 30*, *chunk_size: int = 65536*, *seek: bool = True*) → BinaryIO | None

Download file by file_id or Downloadable object to destination.

If you want to automatically create destination (`io.BytesIO`) use default value of destination and handle result of this method.

Parameters

- **file** – file_id or Downloadable object
- **destination** – Filename, file path or instance of `io.IOBase`. For e.g. `io.BytesIO`, defaults to None
- **timeout** – Total timeout in seconds, defaults to 30
- **chunk_size** – File chunks size, defaults to 64 kb
- **seek** – Go to start of file when downloading is finished. Used only for destination with `typing.BinaryIO` type, defaults to True

It differs from *download_file* **only** in that it accepts *file_id* or an *Downloadable* object (object that contains the *file_id* attribute) instead of *file_path*.

You can download a file to *disk* or to a *binary I/O* object in the same way.

Example:

```
document = message.document
await bot.download(document)
```

2.3.7 How to upload file?

As says official Telegram Bot API documentation there are three ways to send files (photos, stickers, audio, media, etc.):

If the file is already stored somewhere on the Telegram servers or file is available by the URL, you don't need to reupload it.

But if you need to upload a new file just use subclasses of InputFile.

Here are the three different available builtin types of input file:

- `aiogram.types.input_file.FSInputFile` - *uploading from file system*
- `aiogram.types.input_file.BufferedInputFile` - *uploading from buffer*
- `aiogram.types.input_file.URLInputFile` - *uploading from URL*

Warning

Be respectful to Telegram

Instances of *InputFile* are reusable. That means you can create an instance of InputFile and send it multiple times. However, Telegram does not recommend doing this. Instead, once you upload a file, save its *file_id* and reuse that later.

Upload from file system

By first step you will need to import InputFile wrapper:

```
from aiogram.types import FSInputFile
```

Then you can use it:

```
cat = FSInputFile("cat.png")
agenda = FSInputFile("my-document.pdf", filename="agenda-2019-11-19.pdf")
```

class `aiogram.types.input_file.`**`FSInputFile`**(*path: str | Path, filename: str | None = None, chunk_size: int = 65536*)

`__init__`(*path: str | Path, filename: str | None = None, chunk_size: int = 65536*)

Represents object for uploading files from filesystem

Parameters

- **`path`** – Path to file

- **filename** – Filename to be propagated to telegram. By default, will be parsed from path
- **chunk_size** – Uploading chunk size

Upload from buffer

Files can be also passed from buffer (For example you generate image using Pillow and you want to send it to Telegram):

Import wrapper:

```
from aiogram.types import BufferedInputFile
```

And then you can use it:

```
text_file = BufferedInputFile(b"Hello, world!", filename="file.txt")
```

class aiogram.types.input_file.**BufferedInputFile**(*file: bytes*, *filename: str*, *chunk_size: int = 65536*)

__init__(*file: bytes*, *filename: str*, *chunk_size: int = 65536*)

Represents object for uploading files from filesystem

Parameters

- **file** – Bytes
- **filename** – Filename to be propagated to telegram.
- **chunk_size** – Uploading chunk size

Upload from url

If you need to upload a file from another server, but the direct link is bound to your server's IP, or you want to bypass native upload limits by URL, you can use *aiogram.types.input_file.URLInputFile*.

Import wrapper:

```
from aiogram.types import URLInputFile
```

And then you can use it:

```
image = URLInputFile(
    "https://www.python.org/static/community_logos/python-powered-h-140x182.png",
    filename="python-logo.png"
)
```

class aiogram.types.input_file.**URLInputFile**(*url: str*, *headers: Dict[str, Any] | None = None*, *filename: str | None = None*, *chunk_size: int = 65536*, *timeout: int = 30*, *bot: 'Bot' | None = None*)

2.3.8 Global defaults

aiogram provides mechanism to set some global defaults for all requests to Telegram Bot API in your application using `aiogram.client.default.DefaultBotProperties` class.

There are some properties that can be set:

class `aiogram.client.default.`**`DefaultBotProperties`**(*, *parse_mode: str | None = None, disable_notification: bool | None = None, protect_content: bool | None = None, allow_sending_without_reply: bool | None = None, link_preview:* LinkPreviewOptions *| None = None, link_preview_is_disabled: bool | None = None, link_preview_prefer_small_media: bool | None = None, link_preview_prefer_large_media: bool | None = None, link_preview_show_above_text: bool | None = None, show_caption_above_media: bool | None = None*)

Default bot properties.

`parse_mode: str | None`

Default parse mode for messages.

`disable_notification: bool | None`

Sends the message silently. Users will receive a notification with no sound.

`protect_content: bool | None`

Protects content from copying.

`allow_sending_without_reply: bool | None`

Allows to send messages without reply.

`link_preview:` *`LinkPreviewOptions`* **`| None`**

Link preview settings.

`link_preview_is_disabled: bool | None`

Disables link preview.

`link_preview_prefer_small_media: bool | None`

Prefer small media in link preview.

`link_preview_prefer_large_media: bool | None`

Prefer large media in link preview.

`link_preview_show_above_text: bool | None`

Show link preview above text.

`show_caption_above_media: bool | None`

Show caption above media.

Note

If you need to override default properties for some requests, you should use *aiogram.client.default.DefaultBotProperties* only for properties that you want to set as defaults and pass explicit values for other properties.

Danger

If you upgrading from aiogram 3.0-3.6 to 3.7, you should update your code to use *aiogram.client.default.DefaultBotProperties*.

Example

Here is an example of setting default parse mode for all requests to Telegram Bot API:

```
bot = Bot(
    token=...,
    default=DefaultBotProperties(
        parse_mode=ParseMode.HTML,
    )
)
```

In this case all messages sent by this bot will be parsed as HTML, so you don't need to specify *parse_mode* in every message you send.

Instead of

```
await bot.send_message(chat_id, text, parse_mode=ParseMode.HTML)
```

you can use

```
await bot.send_message(chat_id, text)
```

and the message will be sent with HTML parse mode.

In some cases you may want to override default properties for some requests. You can do it by passing explicit values to the method:

```
await bot.send_message(chat_id, text, parse_mode=ParseMode.MARKDOWN_V2)
```

In this case the message will be sent with Markdown parse mode instead of default HTML.

Another example of overriding default properties:

```
await bot.send_message(chat_id, text, parse_mode=None)
```

In this case the message will be send withoout parse mode, even if default parse mode is set it may be useful if you want to send message with plain text or aiogram.types.message_entity.MessageEntity.

```
await bot.send_message(
    chat_id=chat_id,
    text=text,
    entities=[MessageEntity(type='bold', offset=0, length=4)],
    parse_mode=None
)
```

2.4 Handling events

aiogram includes Dispatcher mechanism. Dispatcher is needed for handling incoming updates from Telegram.

With dispatcher you can do:

- Handle incoming updates;
- Filter incoming events before it will be processed by specific handler;
- Modify event and related data in middlewares;
- Separate bot functionality between different handlers, modules and packages

Dispatcher is also separated into two entities - Router and Dispatcher. Dispatcher is subclass of router and should be always is root router.

Telegram supports two ways of receiving updates:

- *Webhook* - you should configure your web server to receive updates from Telegram;
- *Long polling* - you should request updates from Telegram.

So, you can use both of them with *aiogram*.

2.4.1 Router

Usage:

```
from aiogram import Router
from aiogram.types import Message

my_router = Router(name=__name__)

@my_router.message()
async def message_handler(message: Message) -> Any:
    await message.answer('Hello from my router!')
```

class aiogram.dispatcher.router.**Router**(**, name: str | None = None*)

Bases: `object`

Router can route update, and it nested update types like messages, callback query, polls and all other event types.

Event handlers can be registered in observer by two ways:

- By observer method - `router.<event_type>.register(handler, <filters, ...>)`
- By decorator - `@router.<event_type>(<filters, ...>)`

`__init__`(**, name: str | None = None*) → None

Parameters
name – Optional router name, can be useful for debugging

`include_router`(*router: Router*) → *Router*

Attach another router.

Parameters
router

Returns

include_routers(**routers: Router*) → None

Attach multiple routers.

Parameters
routers

Returns

resolve_used_update_types(*skip_events: Set[str] | None = None*) → List[str]

Resolve registered event names

Is useful for getting updates only for registered event types.

Parameters
skip_events – skip specified event names

Returns
set of registered names

Event observers

Warning

All handlers always should be asynchronous. The name of the handler function is not important. The event argument name is also not important but it is recommended to not overlap the name with contextual data in due to function can not accept two arguments with the same name.

Here is the list of available observers and examples of how to register handlers

In these examples only decorator-style registering handlers are used, but if you don't like @decorators just use `<event type>.register(...)` method instead.

Message

Attention

Be attentive with filtering this event

You should expect that this event can be with different sets of attributes in different cases

(For example text, sticker and document are always of different content types of message)

Recommended way to check field availability before usage, for example via *magic filter*: `F.text` to handle text, `F.sticker` to handle stickers only and etc.

```python
@router.message()
async def message_handler(message: types.Message) -> Any: pass
```

Edited message

```
@router.edited_message()
async def edited_message_handler(edited_message: types.Message) -> Any: pass
```

Channel post

```
@router.channel_post()
async def channel_post_handler(channel_post: types.Message) -> Any: pass
```

Edited channel post

```
@router.edited_channel_post()
async def edited_channel_post_handler(edited_channel_post: types.Message) -> Any: pass
```

Inline query

```
@router.inline_query()
async def inline_query_handler(inline_query: types.InlineQuery) -> Any: pass
```

Chosen inline query

```
@router.chosen_inline_result()
async def chosen_inline_result_handler(chosen_inline_result: types.ChosenInlineResult) ->
↪ Any: pass
```

Callback query

```
@router.callback_query()
async def callback_query_handler(callback_query: types.CallbackQuery) -> Any: pass
```

Shipping query

```
@router.shipping_query()
async def shipping_query_handler(shipping_query: types.ShippingQuery) -> Any: pass
```

Pre checkout query

```
@router.pre_checkout_query()
async def pre_checkout_query_handler(pre_checkout_query: types.PreCheckoutQuery) -> Any:␣
↪pass
```

Poll

```
@router.poll()
async def poll_handler(poll: types.Poll) -> Any: pass
```

Poll answer

```
@router.poll_answer()
async def poll_answer_handler(poll_answer: types.PollAnswer) -> Any: pass
```

My chat member

```
@router.my_chat_member()
async def my_chat_member_handler(my_chat_member: types.ChatMemberUpdated) -> Any: pass
```

Chat member

```
@router.chat_member()
async def chat_member_handler(chat_member: types.ChatMemberUpdated) -> Any: pass
```

Chat join request

```
@router.chat_join_request()
async def chat_join_request_handler(chat_join_request: types.ChatJoinRequest) -> Any:␣
↪pass
```

Message reaction

```
@router.message_reaction()
async def message_reaction_handler(message_reaction: types.MessageReactionUpdated) ->␣
↪Any: pass
```

Message reaction count

```
@router.message_reaction_count()
async def message_reaction_count_handler(message_reaction_count: types.
↪MessageReactionCountUpdated) -> Any: pass
```

Chat boost

```
@router.chat_boost()
async def chat_boost_handler(chat_boost: types.ChatBoostUpdated) -> Any: pass
```

Remove chat boost

```
@router.removed_chat_boost()
async def removed_chat_boost_handler(removed_chat_boost: types.ChatBoostRemoved) -> Any:␣
↪pass
```

Errors

```
@router.errors()
async def error_handler(exception: types.ErrorEvent) -> Any: pass
```

Is useful for handling errors from other handlers, error event described *here*

Nested routers

Warning

Routers by the way can be nested to an another routers with some limitations:
1. Router **CAN NOT** include itself 1. Routers **CAN NOT** be used for circular including (router 1 include router 2, router 2 include router 3, router 3 include router 1)

Example:

Listing 1: module_1.py

```
```

name
module_1

router2 = Router()

@router2.message() …

Listing 2: module_2.py

```

```

name
module_2

from module_2 import router2

router1 = Router() router1.include_router(router2)

Update

```
@dispatcher.update()
async def message_handler(update: types.Update) -> Any: pass
```

Warning

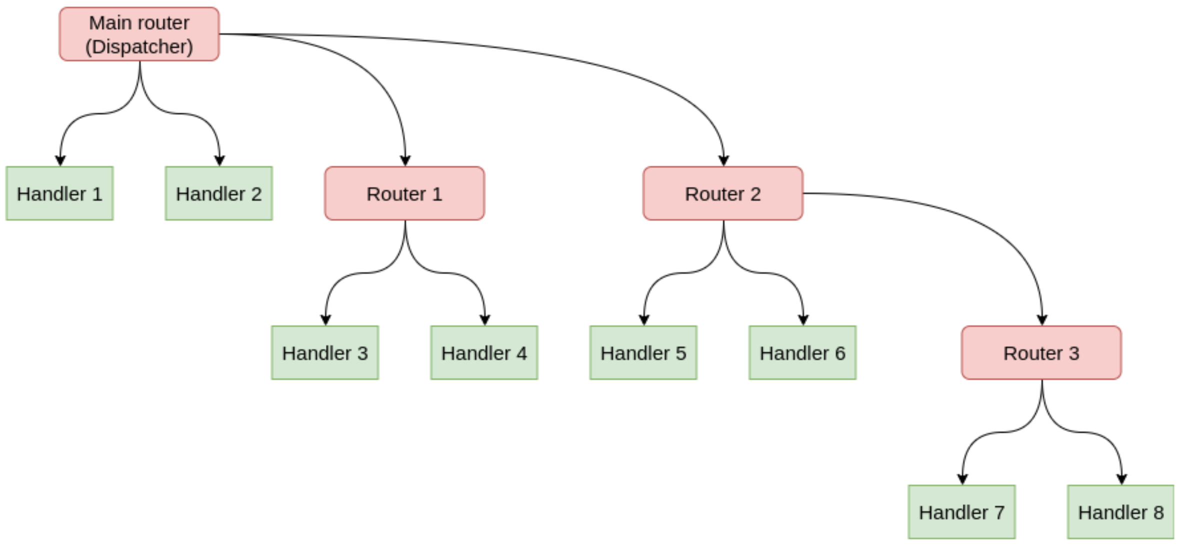
The only root Router (Dispatcher) can handle this type of event.

Note

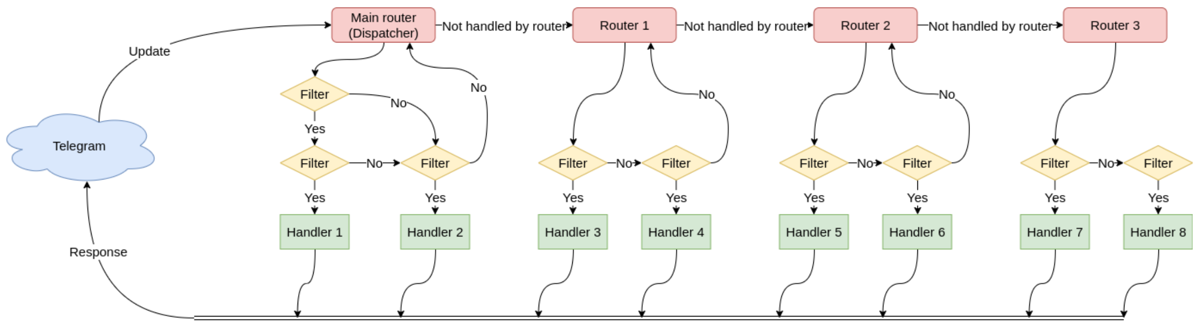
Dispatcher already has default handler for this event type, so you can use it for handling all updates that are not handled by any other handlers.

How it works?

For example, dispatcher has 2 routers, the last router also has one nested router:



In this case update propagation flow will have form:



2.4.2 Dispatcher

Dispatcher is root `Router` and in code Dispatcher can be used directly for routing updates or attach another routers into dispatcher.

Here is only listed base information about Dispatcher. All about writing handlers, filters and etc. you can find in next pages:

- *Router*
- *Filtering events*

class aiogram.dispatcher.dispatcher.**Dispatcher**(*, *storage: BaseStorage | None = None*, *fsm_strategy: FSMStrategy = FSMStrategy.USER_IN_CHAT*, *events_isolation: BaseEventIsolation | None = None*, *disable_fsm: bool = False*, *name: str | None = None*, ***kwargs: Any*)

Root router

__init__(*, *storage: BaseStorage | None = None*, *fsm_strategy: FSMStrategy = FSMStrategy.USER_IN_CHAT*, *events_isolation: BaseEventIsolation | None = None*, *disable_fsm: bool = False*, *name: str | None = None*, ***kwargs: Any*) → None

Root router

Parameters

- **storage** – Storage for FSM
- **fsm_strategy** – FSM strategy
- **events_isolation** – Events isolation
- **disable_fsm** – Disable FSM, note that if you disable FSM then you should not use storage and events isolation
- **kwargs** – Other arguments, will be passed as keyword arguments to handlers

async feed_raw_update(*bot: Bot*, *update: Dict[str, Any]*, ***kwargs: Any*) → Any

Main entry point for incoming updates with automatic Dict->Update serializer

Parameters

- **bot**
- **update**

- **kwargs**

`async feed_update`(*bot: Bot*, *update:* Update, ***kwargs: Any*) → Any

Main entry point for incoming updates Response of this method can be used as Webhook response

Parameters

- **bot**
- **update**

`run_polling`(**bots: Bot*, *polling_timeout: int = 10*, *handle_as_tasks: bool = True*, *backoff_config: BackoffConfig = BackoffConfig(min_delay=1.0, max_delay=5.0, factor=1.3, jitter=0.1)*, *allowed_updates: List[str] | _SentinelObject | None = sentinel.UNSET*, *handle_signals: bool = True*, *close_bot_session: bool = True*, ***kwargs: Any*) → None

Run many bots with polling

Parameters

- **bots** – Bot instances (one or more)
- **polling_timeout** – Long-polling wait time
- **handle_as_tasks** – Run task for each event and no wait result
- **backoff_config** – backoff-retry config
- **allowed_updates** – List of the update types you want your bot to receive
- **handle_signals** – handle signals (SIGINT/SIGTERM)
- **close_bot_session** – close bot sessions on shutdown
- **kwargs** – contextual data

Returns

`async start_polling`(**bots: Bot*, *polling_timeout: int = 10*, *handle_as_tasks: bool = True*, *backoff_config: BackoffConfig = BackoffConfig(min_delay=1.0, max_delay=5.0, factor=1.3, jitter=0.1)*, *allowed_updates: List[str] | _SentinelObject | None = sentinel.UNSET*, *handle_signals: bool = True*, *close_bot_session: bool = True*, ***kwargs: Any*) → None

Polling runner

Parameters

- **bots** – Bot instances (one or more)
- **polling_timeout** – Long-polling wait time
- **handle_as_tasks** – Run task for each event and no wait result
- **backoff_config** – backoff-retry config
- **allowed_updates** – List of the update types you want your bot to receive By default, all used update types are enabled (resolved from handlers)
- **handle_signals** – handle signals (SIGINT/SIGTERM)
- **close_bot_session** – close bot sessions on shutdown
- **kwargs** – contextual data

Returns

async stop_polling() → None

Execute this method if you want to stop polling programmatically

Returns

Simple usage

Example:

```
dp = Dispatcher()

@dp.message()
async def message_handler(message: types.Message) -> None:
    await SendMessage(chat_id=message.from_user.id, text=message.text)
```

Including routers

Example:

```
dp = Dispatcher()
router1 = Router()
dp.include_router(router1)
```

Handling updates

All updates can be propagated to the dispatcher by *feed_update()* method:

```
from aiogram import Bot, Dispatcher

async def update_handler(update: Update, bot: Bot, dispatcher: Dispatcher):
  result = await dp.feed_update(bot, update)
```

Also you can feed raw update (dictionary) object to the dispatcher by *feed_raw_update()* method:

```
from aiogram import Bot, Dispatcher

async def update_handler(raw_update: dict[str, Any], bot: Bot, dispatcher: Dispatcher):
  result = await dp.feed_raw_update(bot, raw_update)
```

2.4.3 Dependency injection

Dependency injection is a programming technique that makes a class independent of its dependencies. It achieves that by decoupling the usage of an object from its creation. This helps you to follow SOLID's dependency inversion and single responsibility principles.

How it works in aiogram

For each update *aiogram.dispatcher.dispatcher.Dispatcher* passes handling context data. Filters and middleware can also make changes to the context.

To access contextual data you should specify corresponding keyword parameter in handler or filter. For example, to get aiogram.fsm.context.FSMContext we do it like that:

```python
@router.message(ProfileCompletion.add_photo, F.photo)
async def add_photo(
    message: types.Message, bot: Bot, state: FSMContext
) -> Any:
    ... # do something with photo
```

Injecting own dependencies

Aiogram provides several ways to complement / modify contextual data.

The first and easiest way is to simply specify the named arguments in *aiogram.dispatcher.dispatcher.Dispatcher* initialization, polling start methods or *aiogram.webhook.aiohttp_server.SimpleRequestHandler* initialization if you use webhooks.

```python
async def main() -> None:
    dp = Dispatcher(..., foo=42)
    return await dp.start_polling(
        bot, bar="Bazz"
    )
```

Analogy for webhook:

```python
async def main() -> None:
    dp = Dispatcher(..., foo=42)
    handler = SimpleRequestHandler(dispatcher=dp, bot=bot, bar="Bazz")
    ... # starting webhook
```

aiogram.dispatcher.dispatcher.Dispatcher's workflow data also can be supplemented by setting values as in a dictionary:

```python
dp = Dispatcher(...)
dp["eggs"] = Spam()
```

The middlewares updates the context quite often. You can read more about them on this page:

- *Middlewares*

The last way is to return a dictionary from the filter:

```python
from typing import Any, Dict, Optional, Union

from aiogram import Router
from aiogram.filters import Filter
from aiogram.types import Message, User

router = Router(name=__name__)

```

(continues on next page)

(continued from previous page)

```
class HelloFilter(Filter):
    def __init__(self, name: Optional[str] = None) -> None:
        self.name = name

    async def __call__(
        self,
        message: Message,
        event_from_user: User,
        # Filters also can accept keyword parameters like in handlers
    ) -> Union[bool, Dict[str, Any]]:
        if message.text.casefold() == "hello":
            # Returning a dictionary that will update the context data
            return {"name": event_from_user.mention_html(name=self.name)}
        return False


@router.message(HelloFilter())
async def my_handler(
    message: Message, name: str  # Now we can accept "name" as named parameter
) -> Any:
    return message.answer("Hello, {name}!".format(name=name))
```

…or using *MagicFilter* with `.as_(...)` method.

2.4.4 Filtering events

Filters is needed for routing updates to the specific handler. Searching of handler is always stops on first match set of filters are pass. By default, all handlers has empty set of filters, so all updates will be passed to first handler that has empty set of filters.

aiogram has some builtin useful filters or you can write own filters.

Builtin filters

Here is list of builtin filters:

Command

Usage

1. Filter single variant of commands: `Command("start")`
2. Handle command by regexp pattern: `Command(re.compile(r"item_(\d+)"))`
3. Match command by multiple variants: `Command("item", re.compile(r"item_(\d+)"))`
4. Handle commands in public chats intended for other bots: `Command("command", ignore_mention=True)`
5. Use *`aiogram.types.bot_command.BotCommand`* object as command reference `Command(BotCommand(command="command", description="My awesome command")`

Warning

Command cannot include spaces or any whitespace

class aiogram.filters.command.**Command**(**values: str | Pattern | BotCommand*, *commands: Sequence[str | Pattern | BotCommand] | str | Pattern | BotCommand | None = None*, *prefix: str = '/'*, *ignore_case: bool = False*, *ignore_mention: bool = False*, *magic: MagicFilter | None = None*)

This filter can be helpful for handling commands from the text messages.

Works only with `aiogram.types.message.Message` events which have the `text`.

__init__(**values: str | Pattern | BotCommand*, *commands: Sequence[str | Pattern | BotCommand] | str | Pattern | BotCommand | None = None*, *prefix: str = '/'*, *ignore_case: bool = False*, *ignore_mention: bool = False*, *magic: MagicFilter | None = None*)

List of commands (string or compiled regexp patterns)

Parameters

- **prefix** – Prefix for command. Prefix is always a single char but here you can pass all of allowed prefixes, for example: `"/!"` will work with commands prefixed by `"/"` or `"!"`.
- **ignore_case** – Ignore case (Does not work with regexp, use flags instead)
- **ignore_mention** – Ignore bot mention. By default, bot can not handle commands intended for other bots
- **magic** – Validate command object via Magic filter after all checks done

When filter is passed the `aiogram.filters.command.CommandObject` will be passed to the handler argument `command`

class aiogram.filters.command.**CommandObject**(*prefix: str = '/'*, *command: str = ''*, *mention: str | None = None*, *args: str | None = None*, *regexp_match: Match[str] | None = None*, *magic_result: Any | None = None*)

Instance of this object is always has command and it prefix. Can be passed as keyword argument **command** to the handler

prefix: str = '/'

Command prefix

command: str = ''

Command without prefix and mention

mention: str | None = None

Mention (if available)

args: str | None = None

Command argument

regexp_match: Match[str] | None = None

Will be presented match result if the command is presented as regexp in filter

magic_result: Any | None = None

property mentioned: bool

This command has mention?

property text: str

Generate original text from object

Allowed handlers

Allowed update types for this filter:

- *message*
- *edited_message*

ChatMemberUpdated

Usage

Handle user leave or join events

```
from aiogram.filters import IS_MEMBER, IS_NOT_MEMBER

@router.chat_member(ChatMemberUpdatedFilter(IS_MEMBER >> IS_NOT_MEMBER))
async def on_user_leave(event: ChatMemberUpdated): ...

@router.chat_member(ChatMemberUpdatedFilter(IS_NOT_MEMBER >> IS_MEMBER))
async def on_user_join(event: ChatMemberUpdated): ...
```

Or construct your own terms via using pre-defined set of statuses and transitions.

Explanation

class aiogram.filters.chat_member_updated.**ChatMemberUpdatedFilter**(*member_status_changed: _MemberStatusMarker | _MemberStatusGroupMarker | _MemberStatusTransition*)

member_status_changed

You can import from `aiogram.filters` all available variants of *statuses*, *status groups* or *transitions*:

Statuses

name	Description
`CREATOR`	Chat owner
`ADMINISTRATOR`	Chat administrator
`MEMBER`	Member of the chat
`RESTRICTED`	Restricted user (can be not member)
`LEFT`	Isn't member of the chat
`KICKED`	Kicked member by administrators

Statuses can be extended with *is_member* flag by prefixing with + (for `is_member == True`) or - (for `is_member == False`) symbol, like `+RESTRICTED` or `-RESTRICTED`

Status groups

The particular statuses can be combined via bitwise `or` operator, like `CREATOR | ADMINISTRATOR`

name	Description
`IS_MEMBER`	Combination of `(CREATOR | ADMINISTRATOR | MEMBER | +RESTRICTED)` statuses.
`IS_ADMIN`	Combination of `(CREATOR | ADMINISTRATOR)` statuses.
`IS_NOT_MEMBER`	Combination of `(LEFT | KICKED | -RESTRICTED)` statuses.

Transitions

Transitions can be defined via bitwise shift operators >> and <<. Old chat member status should be defined in the left side for >> operator (right side for <<) and new status should be specified on the right side for >> operator (left side for <<)

The direction of transition can be changed via bitwise inversion operator: `~JOIN_TRANSITION` will produce swap of old and new statuses.

name	Description
`JOIN_TRANSIT`	Means status changed from `IS_NOT_MEMBER` to `IS_MEMBER` (`IS_NOT_MEMBER >> IS_MEMBER`)
`LEAVE_TRANSI`	Means status changed from `IS_MEMBER` to `IS_NOT_MEMBER` (`~JOIN_TRANSITION`)
`PROMOTED_TRA`	Means status changed from `(MEMBER | RESTRICTED | LEFT | KICKED) >> ADMINISTRATOR` (`(MEMBER | RESTRICTED | LEFT | KICKED) >> ADMINISTRATOR`)

Note

Note that if you define the status unions (via `|`) you will need to add brackets for the statement before use shift operator in due to operator priorities.

Allowed handlers

Allowed update types for this filter:

- *my_chat_member*
- *chat_member*

Magic filters

Note

This page still in progress. Has many incorrectly worded sentences.

Is external package maintained by *aiogram* core team.

By default installs with *aiogram* and also is available on PyPi - magic-filter. That's mean you can install it and use with any other libraries and in own projects without depending *aiogram* installed.

Usage

The **magic_filter** package implements class shortly named `magic_filter.F` that's mean `F` can be imported from `aiogram` or `magic_filter`. `F` is alias for `MagicFilter`.

Note

Note that *aiogram* has an small extension over magic-filter and if you want to use this extension you should import magic from *aiogram* instead of *magic_filter* package

The `MagicFilter` object is callable, supports *some actions* and memorize the attributes chain and the action which should be checked on demand.

So that's mean you can chain attribute getters, describe simple data validations and then call the resulted object passing single object as argument, for example make attributes chain `F.foo.bar.baz` then add action '`F.foo.bar.baz == 'spam'` and then call the resulted object - `(F.foo.bar.baz == 'spam').resolve(obj)`

Possible actions

Magic filter object supports some of basic logical operations over object attributes

Exists or not None

Default actions.

```
F.photo  # lambda message: message.photo
```

Equals

```
F.text == 'hello'  # lambda message: message.text == 'hello'
F.from_user.id == 42  # lambda message: message.from_user.id == 42
F.text != 'spam'  # lambda message: message.text != 'spam'
```

Is one of

Can be used as method named `in_` or as matmul operator `@` with any iterable

```
F.from_user.id.in_({42, 1000, 123123})  # lambda query: query.from_user.id in {42, 1000,␣
↪123123}
F.data.in_({'foo', 'bar', 'baz'})  # lambda query: query.data in {'foo', 'bar', 'baz'}
```

Contains

```
F.text.contains('foo')  # lambda message: 'foo' in message.text
```

String startswith/endswith

Can be applied only for text attributes

```
F.text.startswith('foo')  # lambda message: message.text.startswith('foo')
F.text.endswith('bar')  # lambda message: message.text.startswith('bar')
```

Regexp

```
F.text.regexp(r'Hello, .+')  # lambda message: re.match(r'Hello, .+', message.text)
```

Custom function

Accepts any callable. Callback will be called when filter checks result

```
F.chat.func(lambda chat: chat.id == -42)  # lambda message: (lambda chat: chat.id == -
↪42)(message.chat)
```

Inverting result

Any of available operation can be inverted by bitwise inversion - ~

```
~F.text  # lambda message: not message.text
~F.text.startswith('spam')  # lambda message: not message.text.startswith('spam')
```

Combining

All operations can be combined via bitwise and/or operators - &/ |

```
(F.from_user.id == 42) & (F.text == 'admin')
F.text.startswith('a') | F.text.endswith('b')
(F.from_user.id.in_({42, 777, 911})) & (F.text.startswith('!') | F.text.startswith('/'))␣
↪& F.text.contains('ban')
```

Attribute modifiers - string manipulations

Make text upper- or lower-case

Can be used only with string attributes.

```
F.text.lower() == 'test'  # lambda message: message.text.lower() == 'test'
F.text.upper().in_({'FOO', 'BAR'})  # lambda message: message.text.upper() in {'FOO', 'BAR'}
F.text.len() == 5  # lambda message: len(message.text) == 5
```

Get filter result as handler argument

This part is not available in *magic-filter* directly but can be used with *aiogram*

```
from aiogram import F

...

@router.message(F.text.regexp(r"^(\d+)$").as_("digits"))
async def any_digits_handler(message: Message, digits: Match[str]):
    await message.answer(html.quote(str(digits)))
```

Usage in *aiogram*

```
@router.message(F.text == 'hello')
@router.inline_query(F.data == 'button:1')
@router.message(F.text.startswith('foo'))
@router.message(F.content_type.in_({'text', 'sticker'}))
@router.message(F.text.regexp(r'\d+'))

...

# Many others cases when you will need to check any of available event attribute
```

MagicData

Usage

1. `MagicData(F.event.from_user.id == F.config.admin_id)` (Note that `config` should be passed from middleware)

Explanation

class `aiogram.filters.magic_data.`**`MagicData`**(*magic_data: MagicFilter*)

This filter helps to filter event with contextual data

`magic_data`

Can be imported:

- `from aiogram.filters import MagicData`

Allowed handlers

Allowed update types for this filter:

- `message`
- `edited_message`
- `channel_post`
- `edited_channel_post`
- `inline_query`
- `chosen_inline_result`
- `callback_query`
- `shipping_query`
- `pre_checkout_query`
- `poll`
- `poll_answer`
- `my_chat_member`
- `chat_member`
- `chat_join_request`
- `error`

Callback Data Factory & Filter

class `aiogram.filters.callback_data.`**`CallbackData`**

Base class for callback data wrapper

This class should be used as super-class of user-defined callbacks.

The class-keyword `prefix` is required to define prefix and also the argument `sep` can be passed to define separator (default is `:`).

`pack`() → str

Generate callback data string

Returns

valid callback data for Telegram Bot API

classmethod unpack(*value: str*) → T

Parse callback data string

Parameters
value – value from Telegram

Returns
instance of CallbackData

classmethod filter(*rule: MagicFilter | None = None*) → CallbackQueryFilter

Generates a filter for callback query with rule

Parameters
rule – magic rule

Returns
instance of filter

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

Usage

Create subclass of `CallbackData`:

```
class MyCallback(CallbackData, prefix="my"):
    foo: str
    bar: int
```

After that you can generate any callback based on this class, for example:

```
cb1 = MyCallback(foo="demo", bar=42)
cb1.pack()  # returns 'my:demo:42'
cb1.unpack('my:demo:42')  # returns <MyCallback(foo="demo", bar=42)>
```

So… Now you can use this class to generate any callbacks with defined structure

```
...
# Pass it into the markup
InlineKeyboardButton(
    text="demo",
    callback_data=MyCallback(foo="demo", bar="42").pack()  # value should be packed to␣
↪string
)
...
```

… and handle by specific rules

```
# Filter callback by type and value of field :code:`foo`
@router.callback_query(MyCallback.filter(F.foo == "demo"))
async def my_callback_foo(query: CallbackQuery, callback_data: MyCallback):
    await query.answer(...)
    ...
    print("bar =", callback_data.bar)
```

Also can be used in *Keyboard builder*:

```python
builder = InlineKeyboardBuilder()
builder.button(
    text="demo",
    callback_data=MyCallback(foo="demo", bar="42")  # Value can be not packed to string␣
↪inplace, because builder knows what to do with callback instance
)
```

Another abstract example:

```python
class Action(str, Enum):
    ban = "ban"
    kick = "kick"
    warn = "warn"

class AdminAction(CallbackData, prefix="adm"):
    action: Action
    chat_id: int
    user_id: int


...
# Inside handler
builder = InlineKeyboardBuilder()
for action in Action:
    builder.button(
        text=action.value.title(),
        callback_data=AdminAction(action=action, chat_id=chat_id, user_id=user_id),
    )
await bot.send_message(
    chat_id=admins_chat,
    text=f"What do you want to do with {html.quote(name)}",
    reply_markup=builder.as_markup(),
)
...

@router.callback_query(AdminAction.filter(F.action == Action.ban))
async def ban_user(query: CallbackQuery, callback_data: AdminAction, bot: Bot):
    await bot.ban_chat_member(
        chat_id=callback_data.chat_id,
        user_id=callback_data.user_id,
        ...
    )
```

Known limitations

Allowed types and their subclasses:

- `str`
- `int`
- `bool`
- `float`
- `Decimal` (`from decimal import Decimal`)

- `Fraction` (`from fractions import Fraction`)
- `UUID` (`from uuid import UUID`)
- `Enum` (`from enum import Enum`, only for string enums)
- `IntEnum` (`from enum import IntEnum`, only for int enums)

Note

Note that the integer Enum's should be always is subclasses of `IntEnum` in due to parsing issues.

Exceptions

This filters can be helpful for handling errors from the text messages.

`class aiogram.filters.exception.`**`ExceptionTypeFilter`**`(`**exceptions: Type[Exception]*`)`

Allows to match exception by type

`exceptions`

`class aiogram.filters.exception.`**`ExceptionMessageFilter`**`(`*pattern: str | Pattern[str]*`)`

Allow to match exception by message

`pattern`

Allowed handlers

Allowed update types for this filters:

- `error`

Writing own filters

Filters can be:

- Asynchronous function (`async def my_filter(*args, **kwargs): pass`)
- Synchronous function (`def my_filter(*args, **kwargs): pass`)
- Anonymous function (`lambda event: True`)
- Any awaitable object
- Subclass of *`aiogram.filters.base.Filter`*
- Instances of *MagicFilter*

and should return bool or dict. If the dictionary is passed as result of filter - resulted data will be propagated to the next filters and handler as keywords arguments.

Base class for own filters

class aiogram.filters.base.**Filter**

If you want to register own filters like builtin filters you will need to write subclass of this class with overriding the `__call__` method and adding filter attributes.

abstract async **`__call__`**(**args: Any*, ***kwargs: Any*) → bool | Dict[str, Any]

This method should be overridden.

Accepts incoming event and should return boolean or dict.

Returns
`bool` or `Dict[str, Any]`

update_handler_flags(*flags: Dict[str, Any]*) → None

Also if you want to extend handler flags with using this filter you should implement this method

Parameters
flags – existing flags, can be updated directly

Own filter example

For example if you need to make simple text filter:

```python
from aiogram import Router
from aiogram.filters import Filter
from aiogram.types import Message

router = Router()


class MyFilter(Filter):
    def __init__(self, my_text: str) -> None:
        self.my_text = my_text

    async def __call__(self, message: Message) -> bool:
        return message.text == self.my_text


@router.message(MyFilter("hello"))
async def my_handler(message: Message): ...
```

Combining Filters

In general, all filters can be combined in two ways

Recommended way

If you specify multiple filters in a row, it will be checked with an “and” condition:

```
@<router>.message(F.text.startswith("show"), F.text.endswith("example"))
```

Also, if you want to use two alternative ways to run the same handler (“or” condition) you can register the handler twice or more times as you like

```
@<router>.message(F.text == "hi")
@<router>.message(CommandStart())
```

Also sometimes you will need to invert the filter result, for example you have an *IsAdmin* filter and you want to check if the user is not an admin

```
@<router>.message(~IsAdmin())
```

Another possible way

An alternative way is to combine using special functions (`and_f()`, `or_f()`, `invert_f()` from `aiogram.filters` module):

```
and_f(F.text.startswith("show"), F.text.endswith("example"))
or_f(F.text(text="hi"), CommandStart())
invert_f(IsAdmin())
and_f(<A>, or_f(<B>, <C>))
```

2.4.5 Long-polling

Long-polling is a technology that allows a Telegram server to send updates in case when you don’t have dedicated IP address or port to receive webhooks for example on a developer machine.

To use long-polling mode you should use *`aiogram.dispatcher.dispatcher.Dispatcher.start_polling()`* or *`aiogram.dispatcher.dispatcher.Dispatcher.run_polling()`* methods.

Note

You can use polling from only one polling process per single Bot token, in other case Telegram server will return an error.

Note

If you will need to scale your bot, you should use webhooks instead of long-polling.

Note

If you will use multibot mode, you should use webhook mode for all bots.

Example

This example will show you how to create simple echo bot based on long-polling.

```python
import asyncio
import logging
import sys
from os import getenv

from aiogram import Bot, Dispatcher, html
from aiogram.client.default import DefaultBotProperties
from aiogram.enums import ParseMode
from aiogram.filters import CommandStart
from aiogram.types import Message

# Bot token can be obtained via https://t.me/BotFather
TOKEN = getenv("BOT_TOKEN")

# All handlers should be attached to the Router (or Dispatcher)

dp = Dispatcher()


@dp.message(CommandStart())
async def command_start_handler(message: Message) -> None:
    """
    This handler receives messages with `/start` command
    """
    # Most event objects have aliases for API methods that can be called in events'␣
↪context
    # For example if you want to answer to incoming message you can use `message.answer(.
↪..)` alias
    # and the target chat will be passed to :ref:`aiogram.methods.send_message.
↪SendMessage`
    # method automatically or call API method directly via
    # Bot instance: `bot.send_message(chat_id=message.chat.id, ...)`
    await message.answer(f"Hello, {html.bold(message.from_user.full_name)}!")


@dp.message()
async def echo_handler(message: Message) -> None:
    """
    Handler will forward receive a message back to the sender

    By default, message handler will handle all message types (like a text, photo,␣
↪sticker etc.)
    """
    try:
        # Send a copy of the received message
        await message.send_copy(chat_id=message.chat.id)
    except TypeError:
        # But not all the types is supported to be copied so need to handle it
        await message.answer("Nice try!")
```

(continues on next page)

(continued from previous page)

```
async def main() -> None:
    # Initialize Bot instance with default bot properties which will be passed to all␣
↪API calls
    bot = Bot(token=TOKEN, default=DefaultBotProperties(parse_mode=ParseMode.HTML))

    # And the run events dispatching
    await dp.start_polling(bot)


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, stream=sys.stdout)
    asyncio.run(main())
```

2.4.6 Finite State Machine

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation.

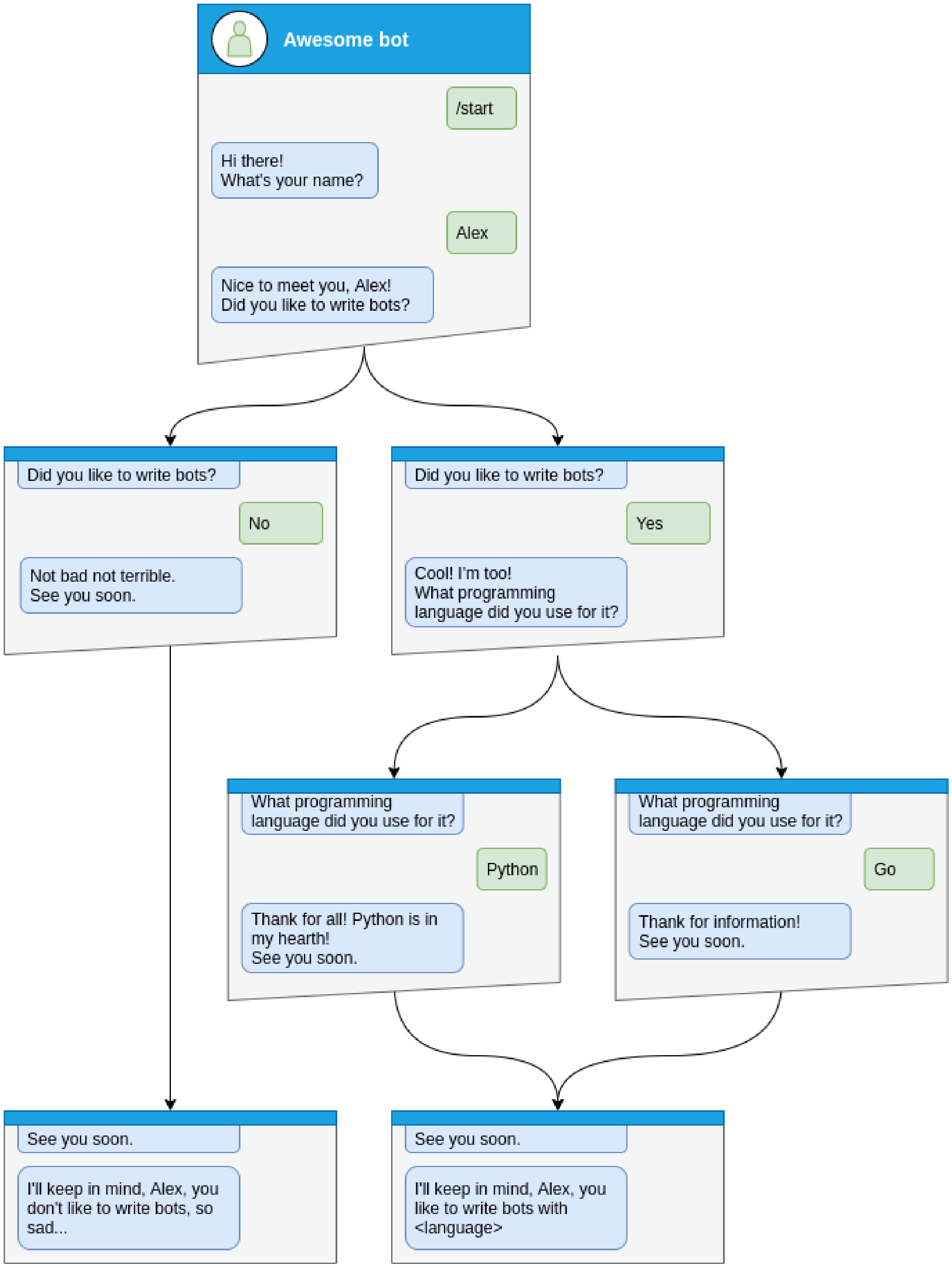
It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition.

An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition.

Source: WikiPedia

Usage example

Not all functionality of the bot can be implemented as single handler, for example you will need to collect some data from user in separated steps you will need to use FSM.



Let's see how to do that step-by-step

Step by step

Before handle any states you will need to specify what kind of states you want to handle

```
class Form(StatesGroup):
    name = State()
    like_bots = State()
    language = State()
```

And then write handler for each state separately from the start of dialog

Here is dialog can be started only via command `/start`, so lets handle it and make transition user to state `Form.name`

```
@form_router.message(CommandStart())
async def command_start(message: Message, state: FSMContext) -> None:
    await state.set_state(Form.name)
    await message.answer(
        "Hi there! What's your name?",
        reply_markup=ReplyKeyboardRemove(),
    )
```

After that you will need to save some data to the storage and make transition to next step.

```
@form_router.message(Form.name)
async def process_name(message: Message, state: FSMContext) -> None:
    await state.update_data(name=message.text)
    await state.set_state(Form.like_bots)
    await message.answer(
        f"Nice to meet you, {html.quote(message.text)}!\nDid you like to write bots?",
        reply_markup=ReplyKeyboardMarkup(
            keyboard=[
                [
                    KeyboardButton(text="Yes"),
                    KeyboardButton(text="No"),
                ]
            ],
            resize_keyboard=True,
        ),
    )
```

At the next steps user can make different answers, it can be *yes*, *no* or any other

Handle `yes` and soon we need to handle `Form.language` state

```
@form_router.message(Form.like_bots, F.text.casefold() == "yes")
async def process_like_write_bots(message: Message, state: FSMContext) -> None:
    await state.set_state(Form.language)

    await message.reply(
        "Cool! I'm too!\nWhat programming language did you use for it?",
        reply_markup=ReplyKeyboardRemove(),
    )
```

Handle `no`

```
@form_router.message(Form.like_bots, F.text.casefold() == "no")
async def process_dont_like_write_bots(message: Message, state: FSMContext) -> None:
    data = await state.get_data()
    await state.clear()
    await message.answer(
        "Not bad not terrible.\nSee you soon.",
        reply_markup=ReplyKeyboardRemove(),
    )
    await show_summary(message=message, data=data, positive=False)
```

And handle any other answers

```
@form_router.message(Form.like_bots)
async def process_unknown_write_bots(message: Message) -> None:
    await message.reply("I don't understand you :(")
```

All possible cases of *like_bots* step was covered, let’s implement finally step

```
@form_router.message(Form.language)
async def process_language(message: Message, state: FSMContext) -> None:
    data = await state.update_data(language=message.text)
    await state.clear()

    if message.text.casefold() == "python":
        await message.reply(
            "Python, you say? That's the language that makes my circuits light up! "
        )

    await show_summary(message=message, data=data)
```

```
async def show_summary(message: Message, data: Dict[str, Any], positive: bool = True) ->␣
↪None:
    name = data["name"]
    language = data.get("language", "<something unexpected>")
    text = f"I'll keep in mind that, {html.quote(name)}, "
    text += (
        f"you like to write bots with {html.quote(language)}."
        if positive
        else "you don't like to write bots, so sad..."
    )
    await message.answer(text=text, reply_markup=ReplyKeyboardRemove())
```

And now you have covered all steps from the image, but you can make possibility to cancel conversation, lets do that via command or text

```
@form_router.message(Command("cancel"))
@form_router.message(F.text.casefold() == "cancel")
async def cancel_handler(message: Message, state: FSMContext) -> None:
    """
    Allow user to cancel any action
    """
    current_state = await state.get_state()
    if current_state is None:
```

(continues on next page)

(continued from previous page)

```
        return

    logging.info("Cancelling state %r", current_state)
    await state.clear()
    await message.answer(
        "Cancelled.",
        reply_markup=ReplyKeyboardRemove(),
    )
```

Complete example

```
import asyncio
import logging
import sys
from os import getenv
from typing import Any, Dict

from aiogram import Bot, Dispatcher, F, Router, html
from aiogram.client.default import DefaultBotProperties
from aiogram.enums import ParseMode
from aiogram.filters import Command, CommandStart
from aiogram.fsm.context import FSMContext
from aiogram.fsm.state import State, StatesGroup
from aiogram.types import (
    KeyboardButton,
    Message,
    ReplyKeyboardMarkup,
    ReplyKeyboardRemove,
)

TOKEN = getenv("BOT_TOKEN")

form_router = Router()


class Form(StatesGroup):
    name = State()
    like_bots = State()
    language = State()


@form_router.message(CommandStart())
async def command_start(message: Message, state: FSMContext) -> None:
    await state.set_state(Form.name)
    await message.answer(
        "Hi there! What's your name?",
        reply_markup=ReplyKeyboardRemove(),
    )


```

(continues on next page)

(continued from previous page)

```python
@form_router.message(Command("cancel"))
@form_router.message(F.text.casefold() == "cancel")
async def cancel_handler(message: Message, state: FSMContext) -> None:
    """
    Allow user to cancel any action
    """
    current_state = await state.get_state()
    if current_state is None:
        return

    logging.info("Cancelling state %r", current_state)
    await state.clear()
    await message.answer(
        "Cancelled.",
        reply_markup=ReplyKeyboardRemove(),
    )


@form_router.message(Form.name)
async def process_name(message: Message, state: FSMContext) -> None:
    await state.update_data(name=message.text)
    await state.set_state(Form.like_bots)
    await message.answer(
        f"Nice to meet you, {html.quote(message.text)}!\nDid you like to write bots?",
        reply_markup=ReplyKeyboardMarkup(
            keyboard=[
                [
                    KeyboardButton(text="Yes"),
                    KeyboardButton(text="No"),
                ]
            ],
            resize_keyboard=True,
        ),
    )


@form_router.message(Form.like_bots, F.text.casefold() == "no")
async def process_dont_like_write_bots(message: Message, state: FSMContext) -> None:
    data = await state.get_data()
    await state.clear()
    await message.answer(
        "Not bad not terrible.\nSee you soon.",
        reply_markup=ReplyKeyboardRemove(),
    )
    await show_summary(message=message, data=data, positive=False)


@form_router.message(Form.like_bots, F.text.casefold() == "yes")
async def process_like_write_bots(message: Message, state: FSMContext) -> None:
    await state.set_state(Form.language)

    await message.reply(
```

(continues on next page)

(continued from previous page)

```python
        "Cool! I'm too!\nWhat programming language did you use for it?",
        reply_markup=ReplyKeyboardRemove(),
    )


@form_router.message(Form.like_bots)
async def process_unknown_write_bots(message: Message) -> None:
    await message.reply("I don't understand you :(")


@form_router.message(Form.language)
async def process_language(message: Message, state: FSMContext) -> None:
    data = await state.update_data(language=message.text)
    await state.clear()

    if message.text.casefold() == "python":
        await message.reply(
            "Python, you say? That's the language that makes my circuits light up! "
        )

    await show_summary(message=message, data=data)


async def show_summary(message: Message, data: Dict[str, Any], positive: bool = True) ->␣
↪None:
    name = data["name"]
    language = data.get("language", "<something unexpected>")
    text = f"I'll keep in mind that, {html.quote(name)}, "
    text += (
        f"you like to write bots with {html.quote(language)}."
        if positive
        else "you don't like to write bots, so sad..."
    )
    await message.answer(text=text, reply_markup=ReplyKeyboardRemove())


async def main():
    # Initialize Bot instance with default bot properties which will be passed to all␣
↪API calls
    bot = Bot(token=TOKEN, default=DefaultBotProperties(parse_mode=ParseMode.HTML))

    dp = Dispatcher()

    dp.include_router(form_router)

    # Start event dispatching
    await dp.start_polling(bot)


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, stream=sys.stdout)
    asyncio.run(main())
```

Read more

Storages

Storages out of the box

MemoryStorage

class `aiogram.fsm.storage.memory.`**`MemoryStorage`**

Default FSM storage, stores all data in `dict` and loss everything on shutdown

> **Warning**
>
> Is not recommended using in production in due to you will lose all data when your bot restarts

`__init__`() → None

RedisStorage

class `aiogram.fsm.storage.redis.`**`RedisStorage`**(*redis: ~redis.asyncio.client.Redis, key_builder: ~aiogram.fsm.storage.base.KeyBuilder | None = None, state_ttl: int | ~datetime.timedelta | None = None, data_ttl: int | ~datetime.timedelta | None = None, json_loads: ~typing.Callable[[...], ~typing.Any] = <function loads>, json_dumps: ~typing.Callable[[...], str] = <function dumps>*)

Redis storage required `redis` package installed (`pip install redis`)

`__init__`(*redis: ~redis.asyncio.client.Redis, key_builder: ~aiogram.fsm.storage.base.KeyBuilder | None = None, state_ttl: int | ~datetime.timedelta | None = None, data_ttl: int | ~datetime.timedelta | None = None, json_loads: ~typing.Callable[[...], ~typing.Any] = <function loads>, json_dumps: ~typing.Callable[[...], str] = <function dumps>*) → None

Parameters

- **redis** – Instance of Redis connection
- **key_builder** – builder that helps to convert contextual key to string
- **state_ttl** – TTL for state records
- **data_ttl** – TTL for data records

classmethod **`from_url`**(*url: str, connection_kwargs: Dict[str, Any] | None = None, **kwargs: Any*) → *RedisStorage*

Create an instance of `RedisStorage` with specifying the connection string

Parameters

- **url** – for example `redis://user:password@host:port/db`
- **connection_kwargs** – see `redis` docs
- **kwargs** – arguments to be passed to `RedisStorage`

Returns
an instance of *RedisStorage*

MongoStorage

class aiogram.fsm.storage.mongo.**MongoStorage**(*client: AsyncIOMotorClient*, *key_builder:* KeyBuilder | *None = None*, *db_name: str = 'aiogram_fsm'*, *collection_name: str = 'states_and_data'*)

MongoDB storage required `motor` package installed (`pip install motor`)

__init__(*client: AsyncIOMotorClient*, *key_builder:* KeyBuilder | *None = None*, *db_name: str = 'aiogram_fsm'*, *collection_name: str = 'states_and_data'*) → None

Parameters

- **client** – Instance of AsyncIOMotorClient
- **key_builder** – builder that helps to convert contextual key to string
- **db_name** – name of the MongoDB database for FSM
- **collection_name** – name of the collection for storing FSM states and data

classmethod from_url(*url: str*, *connection_kwargs: Dict[str, Any] | None = None*, ***kwargs: Any*) → *MongoStorage*

Create an instance of *MongoStorage* with specifying the connection string

Parameters

- **url** – for example `mongodb://user:password@host:port`
- **connection_kwargs** – see `motor` docs
- **kwargs** – arguments to be passed to *MongoStorage*

Returns
an instance of *MongoStorage*

KeyBuilder

Keys inside Redis and Mongo storages can be customized via key builders:

class aiogram.fsm.storage.base.**KeyBuilder**

Base class for key builder.

abstract build(*key: StorageKey*, *part: Literal['data', 'state', 'lock'] | None = None*) → str

Build key to be used in storage's db queries

Parameters

- **key** – contextual key
- **part** – part of the record

Returns
key to be used in storage's db queries

`class aiogram.fsm.storage.base.`**`DefaultKeyBuilder`**`(`**, prefix: str = 'fsm', separator: str = ':', with_bot_id: bool = False, with_business_connection_id: bool = False, with_destiny: bool = False*`)`

Simple key builder with default prefix.

Generates a colon-joined string with prefix, chat_id, user_id, optional bot_id, business_connection_id, destiny and field.

Format:
`<prefix>:<bot_id?>:<business_connection_id?>:<chat_id>:<user_id>:<destiny?>:<field?>`

`build``(`*key: StorageKey, part: Literal['data', 'state', 'lock'] | None = None*`)` → str

Build key to be used in storage's db queries

Parameters

- **`key`** – contextual key
- **`part`** – part of the record

Returns
key to be used in storage's db queries

Writing own storages

`class aiogram.fsm.storage.base.`**`BaseStorage`**

Base class for all FSM storages

`abstract async` **`set_state`**`(`*key: StorageKey, state: str | State | None = None*`)` → None

Set state for specified key

Parameters

- **`key`** – storage key
- **`state`** – new state

`abstract async` **`get_state`**`(`*key: StorageKey*`)` → str | None

Get key state

Parameters
`key` – storage key

Returns
current state

`abstract async` **`set_data`**`(`*key: StorageKey, data: Dict[str, Any]*`)` → None

Write data (replace)

Parameters

- **`key`** – storage key
- **`data`** – new data

`abstract async` **`get_data`**`(`*key: StorageKey*`)` → Dict[str, Any]

Get current data for key

Parameters
`key` – storage key

Returns
current data

async update_data(*key: StorageKey*, *data: Dict[str, Any]*) → Dict[str, Any]

Update date in the storage for key (like dict.update)

Parameters

- `key` – storage key
- `data` – partial data

Returns
new data

abstract async close() → None

Close storage (database connection, file or etc.)

Strategy

This module provides the *FSMStrategy* enumeration which is used to define the strategy of the finite state machine.

class aiogram.fsm.strategy.**FSMStrategy**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

FSM strategy for storage key generation.

CHAT = 2

State will be stored for each chat globally without separating by users.

CHAT_TOPIC = 5

State will be stored for each chat and topic, but not separated by users.

GLOBAL_USER = 3

State will be stored globally for each user globally.

USER_IN_CHAT = 1

State will be stored for each user in chat.

USER_IN_TOPIC = 4

State will be stored for each user in chat and topic.

Scenes Wizard

Added in version 3.2.

Warning

This feature is experimental and may be changed in future versions.

aiogram's basics API is easy to use and powerful, allowing the implementation of simple interactions such as triggering a command or message for a response. However, certain tasks require a dialogue between the user and the bot. This is where Scenes come into play.

Understanding Scenes

A Scene in **aiogram** is like an abstract, isolated namespace or room that a user can be ushered into via the code. When a user is within a Scene, most other global commands or message handlers are bypassed, unless they are specifically designed to function outside of the Scenes. This helps in creating an experience of focused interactions. Scenes provide a structure for more complex interactions, effectively isolating and managing contexts for different stages of the conversation. They allow you to control and manage the flow of the conversation in a more organized manner.

Scene Lifecycle

Each Scene can be "entered", "left" of "exited", allowing for clear transitions between different stages of the conversation. For instance, in a multi-step form filling interaction, each step could be a Scene - the bot guides the user from one Scene to the next as they provide the required information.

Scene Listeners

Scenes have their own hooks which are command or message listeners that only act while the user is within the Scene. These hooks react to user actions while the user is 'inside' the Scene, providing the responses or actions appropriate for that context. When the user is ushered from one Scene to another, the actions and responses change accordingly as the user is now interacting with the set of listeners inside the new Scene. These 'Scene-specific' hooks or listeners, detached from the global listening context, allow for more streamlined and organized bot-user interactions.

Scene Interactions

Each Scene is like a self-contained world, with interactions defined within the scope of that Scene. As such, only the handlers defined within the specific Scene will react to user's input during the lifecycle of that Scene.

Scene Benefits

Scenes can help manage more complex interaction workflows and enable more interactive and dynamic dialogs between the user and the bot. This offers great flexibility in handling multi-step interactions or conversations with the users.

How to use Scenes

For example we have a quiz bot, which asks the user a series of questions and then displays the results.

Lets start with the data models, in this example simple data models are used to represent the questions and answers, in real life you would probably use a database to store the data.

Listing 3: Questions list

```python
@dataclass
class Answer:
    """
    Represents an answer to a question.
    """

    text: str
    """The answer text"""
```

(continues on next page)

(continued from previous page)

```
    is_correct: bool = False
    """Indicates if the answer is correct"""


@dataclass
class Question:
    """
    Class representing a quiz with a question and a list of answers.
    """

    text: str
    """The question text"""
    answers: list[Answer]
    """List of answers"""

    correct_answer: str = field(init=False)

    def __post_init__(self):
        self.correct_answer = next(answer.text for answer in self.answers if answer.is_
↪correct)


# Fake data, in real application you should use a database or something else
QUESTIONS = [
    Question(
        text="What is the capital of France?",
        answers=[
            Answer("Paris", is_correct=True),
            Answer("London"),
            Answer("Berlin"),
            Answer("Madrid"),
        ],
    ),
    Question(
        text="What is the capital of Spain?",
        answers=[
            Answer("Paris"),
            Answer("London"),
            Answer("Berlin"),
            Answer("Madrid", is_correct=True),
        ],
    ),
    Question(
        text="What is the capital of Germany?",
        answers=[
            Answer("Paris"),
            Answer("London"),
            Answer("Berlin", is_correct=True),
            Answer("Madrid"),
        ],
    ),
    Question(
```

(continues on next page)

(continued from previous page)

```
        text="What is the capital of England?",
        answers=[
            Answer("Paris"),
            Answer("London", is_correct=True),
            Answer("Berlin"),
            Answer("Madrid"),
        ],
    ),
    Question(
        text="What is the capital of Italy?",
        answers=[
            Answer("Paris"),
            Answer("London"),
            Answer("Berlin"),
            Answer("Rome", is_correct=True),
        ],
    ),
]
```

Then, we need to create a Scene class that will represent the quiz game scene:

Note

Keyword argument passed into class definition describes the scene name - is the same as state of the scene.

Listing 4: Quiz Scene

```python
class QuizScene(Scene, state="quiz"):
    """
    This class represents a scene for a quiz game.

    It inherits from Scene class and is associated with the state "quiz".
    It handles the logic and flow of the quiz game.
    """
```

Also we need to define a handler that helps to start the quiz game:

Listing 5: Start command handler

```python
quiz_router = Router(name=__name__)
# Add handler that initializes the scene
quiz_router.message.register(QuizScene.as_handler(), Command("quiz"))
```

Once the scene is defined, we need to register it in the SceneRegistry:

Listing 6: Registering the scene

```python
def create_dispatcher():
    # Event isolation is needed to correctly handle fast user responses
    dispatcher = Dispatcher(
        events_isolation=SimpleEventIsolation(),
    )
```

(continues on next page)

(continued from previous page)

```
    dispatcher.include_router(quiz_router)

    # To use scenes, you should create a SceneRegistry and register your scenes there
    scene_registry = SceneRegistry(dispatcher)
    # ... and then register a scene in the registry
    # by default, Scene will be mounted to the router that passed to the SceneRegistry,
    # but you can specify the router explicitly using the `router` argument
    scene_registry.add(QuizScene)

    return dispatcher
```

So, now we can implement the quiz game logic, each question is sent to the user one by one, and the user's answer is checked at the end of all questions.

Now we need to write an entry point for the question handler:

Listing 7: Question handler entry point

```
    @on.message.enter()
    async def on_enter(self, message: Message, state: FSMContext, step: int | None = 0) -
↪> Any:
        """
        Method triggered when the user enters the quiz scene.

        It displays the current question and answer options to the user.

        :param message:
        :param state:
        :param step: Scene argument, can be passed to the scene using the wizard
        :return:
        """
        if not step:
            # This is the first step, so we should greet the user
            await message.answer("Welcome to the quiz!")

        try:
            quiz = QUESTIONS[step]
        except IndexError:
            # This error means that the question's list is over
            return await self.wizard.exit()

        markup = ReplyKeyboardBuilder()
        markup.add(*[KeyboardButton(text=answer.text) for answer in quiz.answers])

        if step > 0:
            markup.button(text=" Back")
        markup.button(text=" Exit")

        await state.update_data(step=step)
        return await message.answer(
            text=QUESTIONS[step].text,
            reply_markup=markup.adjust(2).as_markup(resize_keyboard=True),
        )
```

Once scene is entered, we should expect the user's answer, so we need to write a handler for it, this handler should expect the text message, save the answer and retake the question handler for the next question:

Listing 8: Answer handler

```
    @on.message(F.text)
    async def answer(self, message: Message, state: FSMContext) -> None:
        """
        Method triggered when the user selects an answer.

        It stores the answer and proceeds to the next question.

        :param message:
        :param state:
        :return:
        """
        data = await state.get_data()
        step = data["step"]
        answers = data.get("answers", {})
        answers[step] = message.text
        await state.update_data(answers=answers)

        await self.wizard.retake(step=step + 1)
```

When user answer with unknown message, we should expect the text message again:

Listing 9: Unknown message handler

```
    @on.message()
    async def unknown_message(self, message: Message) -> None:
        """
        Method triggered when the user sends a message that is not a command or an␣
↪answer.

        It asks the user to select an answer.

        :param message: The message received from the user.
        :return: None
        """
        await message.answer("Please select an answer.")
```

When all questions are answered, we should show the results to the user, as you can see in the code below, we use *await self.wizard.exit()* to exit from the scene when questions list is over in the *QuizScene.on_enter* handler.

Thats means that we need to write an exit handler to show the results to the user:

Listing 10: Show results handler

```
    @on.message.exit()
    async def on_exit(self, message: Message, state: FSMContext) -> None:
        """
        Method triggered when the user exits the quiz scene.

        It calculates the user's answers, displays the summary, and clears the stored␣
↪answers.
```

(continues on next page)

(continued from previous page)

```
        :param message:
        :param state:
        :return:
        """
        data = await state.get_data()
        answers = data.get("answers", {})

        correct = 0
        incorrect = 0
        user_answers = []
        for step, quiz in enumerate(QUESTIONS):
            answer = answers.get(step)
            is_correct = answer == quiz.correct_answer
            if is_correct:
                correct += 1
                icon = ""
            else:
                incorrect += 1
                icon = ""
            if answer is None:
                answer = "no answer"
            user_answers.append(f"{quiz.text} ({icon} {html.quote(answer)})")

        content = as_list(
            as_section(
                Bold("Your answers:"),
                as_numbered_list(*user_answers),
            ),
            "",
            as_section(
                Bold("Summary:"),
                as_list(
                    as_key_value("Correct", correct),
                    as_key_value("Incorrect", incorrect),
                ),
            ),
        )

        await message.answer(**content.as_kwargs(), reply_markup=ReplyKeyboardRemove())
        await state.set_data({})
```

Also we can implement a actions to exit from the quiz game or go back to the previous question:

Listing 11: Exit handler

```
    @on.message(F.text == " Exit")
    async def exit(self, message: Message) -> None:
        """
        Method triggered when the user selects the "Exit" button.

        It exits the quiz.
```

(continues on next page)

(continued from previous page)

```
        :param message:
        :return:
        """
        await self.wizard.exit()
```

Listing 12: Back handler

```
    @on.message(F.text == " Back")
    async def back(self, message: Message, state: FSMContext) -> None:
        """
        Method triggered when the user selects the "Back" button.

        It allows the user to go back to the previous question.

        :param message:
        :param state:
        :return:
        """
        data = await state.get_data()
        step = data["step"]

        previous_step = step - 1
        if previous_step < 0:
            # In case when the user tries to go back from the first question,
            # we just exit the quiz
            return await self.wizard.exit()
        return await self.wizard.back(step=previous_step)
```

Now we can run the bot and test the quiz game:

Listing 13: Run the bot

```
async def main():
    dp = create_dispatcher()
    bot = Bot(token=TOKEN)
    await dp.start_polling(bot)


if __name__ == "__main__":
    # Alternatively, you can use aiogram-cli:
    # `aiogram run polling quiz_scene:create_dispatcher --log-level info --token BOT_
↪TOKEN`
    logging.basicConfig(level=logging.INFO)
    asyncio.run(main())
```

Complete them all

Listing 14: Quiz Example

```
import asyncio
import logging
from dataclasses import dataclass, field
```

(continues on next page)

(continued from previous page)

```python
from os import getenv
from typing import Any

from aiogram import Bot, Dispatcher, F, Router, html
from aiogram.filters import Command
from aiogram.fsm.context import FSMContext
from aiogram.fsm.scene import Scene, SceneRegistry, ScenesManager, on
from aiogram.fsm.storage.memory import SimpleEventIsolation
from aiogram.types import KeyboardButton, Message, ReplyKeyboardRemove
from aiogram.utils.formatting import (
    Bold,
    as_key_value,
    as_list,
    as_numbered_list,
    as_section,
)
from aiogram.utils.keyboard import ReplyKeyboardBuilder

TOKEN = getenv("BOT_TOKEN")


@dataclass
class Answer:
    """
    Represents an answer to a question.
    """

    text: str
    """The answer text"""
    is_correct: bool = False
    """Indicates if the answer is correct"""


@dataclass
class Question:
    """
    Class representing a quiz with a question and a list of answers.
    """

    text: str
    """The question text"""
    answers: list[Answer]
    """List of answers"""

    correct_answer: str = field(init=False)

    def __post_init__(self):
        self.correct_answer = next(answer.text for answer in self.answers if answer.is_
↪correct)


# Fake data, in real application you should use a database or something else
```

(continues on next page)

(continued from previous page)

```python
QUESTIONS = [
    Question(
        text="What is the capital of France?",
        answers=[
            Answer("Paris", is_correct=True),
            Answer("London"),
            Answer("Berlin"),
            Answer("Madrid"),
        ],
    ),
    Question(
        text="What is the capital of Spain?",
        answers=[
            Answer("Paris"),
            Answer("London"),
            Answer("Berlin"),
            Answer("Madrid", is_correct=True),
        ],
    ),
    Question(
        text="What is the capital of Germany?",
        answers=[
            Answer("Paris"),
            Answer("London"),
            Answer("Berlin", is_correct=True),
            Answer("Madrid"),
        ],
    ),
    Question(
        text="What is the capital of England?",
        answers=[
            Answer("Paris"),
            Answer("London", is_correct=True),
            Answer("Berlin"),
            Answer("Madrid"),
        ],
    ),
    Question(
        text="What is the capital of Italy?",
        answers=[
            Answer("Paris"),
            Answer("London"),
            Answer("Berlin"),
            Answer("Rome", is_correct=True),
        ],
    ),
]


class QuizScene(Scene, state="quiz"):
    """
    This class represents a scene for a quiz game.
```

(continues on next page)

(continued from previous page)

```
    It inherits from Scene class and is associated with the state "quiz".
    It handles the logic and flow of the quiz game.
    """

    @on.message.enter()
    async def on_enter(self, message: Message, state: FSMContext, step: int | None = 0) -
↪> Any:
        """
        Method triggered when the user enters the quiz scene.

        It displays the current question and answer options to the user.

        :param message:
        :param state:
        :param step: Scene argument, can be passed to the scene using the wizard
        :return:
        """
        if not step:
            # This is the first step, so we should greet the user
            await message.answer("Welcome to the quiz!")

        try:
            quiz = QUESTIONS[step]
        except IndexError:
            # This error means that the question's list is over
            return await self.wizard.exit()

        markup = ReplyKeyboardBuilder()
        markup.add(*[KeyboardButton(text=answer.text) for answer in quiz.answers])

        if step > 0:
            markup.button(text=" Back")
        markup.button(text=" Exit")

        await state.update_data(step=step)
        return await message.answer(
            text=QUESTIONS[step].text,
            reply_markup=markup.adjust(2).as_markup(resize_keyboard=True),
        )

    @on.message.exit()
    async def on_exit(self, message: Message, state: FSMContext) -> None:
        """
        Method triggered when the user exits the quiz scene.

        It calculates the user's answers, displays the summary, and clears the stored␣
↪answers.

        :param message:
        :param state:
        :return:
```

(continues on next page)

(continued from previous page)

```
        """
        data = await state.get_data()
        answers = data.get("answers", {})

        correct = 0
        incorrect = 0
        user_answers = []
        for step, quiz in enumerate(QUESTIONS):
            answer = answers.get(step)
            is_correct = answer == quiz.correct_answer
            if is_correct:
                correct += 1
                icon = ""
            else:
                incorrect += 1
                icon = ""
            if answer is None:
                answer = "no answer"
            user_answers.append(f"{quiz.text} ({icon} {html.quote(answer)})")

        content = as_list(
            as_section(
                Bold("Your answers:"),
                as_numbered_list(*user_answers),
            ),
            "",
            as_section(
                Bold("Summary:"),
                as_list(
                    as_key_value("Correct", correct),
                    as_key_value("Incorrect", incorrect),
                ),
            ),
        )

        await message.answer(**content.as_kwargs(), reply_markup=ReplyKeyboardRemove())
        await state.set_data({})

    @on.message(F.text == " Back")
    async def back(self, message: Message, state: FSMContext) -> None:
        """
        Method triggered when the user selects the "Back" button.

        It allows the user to go back to the previous question.

        :param message:
        :param state:
        :return:
        """
        data = await state.get_data()
        step = data["step"]
```

(continues on next page)

(continued from previous page)

```
        previous_step = step - 1
        if previous_step < 0:
            # In case when the user tries to go back from the first question,
            # we just exit the quiz
            return await self.wizard.exit()
        return await self.wizard.back(step=previous_step)

    @on.message(F.text == " Exit")
    async def exit(self, message: Message) -> None:
        """
        Method triggered when the user selects the "Exit" button.

        It exits the quiz.

        :param message:
        :return:
        """
        await self.wizard.exit()

    @on.message(F.text)
    async def answer(self, message: Message, state: FSMContext) -> None:
        """
        Method triggered when the user selects an answer.

        It stores the answer and proceeds to the next question.

        :param message:
        :param state:
        :return:
        """
        data = await state.get_data()
        step = data["step"]
        answers = data.get("answers", {})
        answers[step] = message.text
        await state.update_data(answers=answers)

        await self.wizard.retake(step=step + 1)

    @on.message()
    async def unknown_message(self, message: Message) -> None:
        """
        Method triggered when the user sends a message that is not a command or an␣
↪answer.

        It asks the user to select an answer.

        :param message: The message received from the user.
        :return: None
        """
        await message.answer("Please select an answer.")
```

(continues on next page)

(continued from previous page)

```python
quiz_router = Router(name=__name__)
# Add handler that initializes the scene
quiz_router.message.register(QuizScene.as_handler(), Command("quiz"))


@quiz_router.message(Command("start"))
async def command_start(message: Message, scenes: ScenesManager):
    await scenes.close()
    await message.answer(
        "Hi! This is a quiz bot. To start the quiz, use the /quiz command.",
        reply_markup=ReplyKeyboardRemove(),
    )


def create_dispatcher():
    # Event isolation is needed to correctly handle fast user responses
    dispatcher = Dispatcher(
        events_isolation=SimpleEventIsolation(),
    )
    dispatcher.include_router(quiz_router)

    # To use scenes, you should create a SceneRegistry and register your scenes there
    scene_registry = SceneRegistry(dispatcher)
    # ... and then register a scene in the registry
    # by default, Scene will be mounted to the router that passed to the SceneRegistry,
    # but you can specify the router explicitly using the `router` argument
    scene_registry.add(QuizScene)

    return dispatcher


async def main():
    dp = create_dispatcher()
    bot = Bot(token=TOKEN)
    await dp.start_polling(bot)


if __name__ == "__main__":
    # Alternatively, you can use aiogram-cli:
    # `aiogram run polling quiz_scene:create_dispatcher --log-level info --token BOT_
↪TOKEN`
    logging.basicConfig(level=logging.INFO)
    asyncio.run(main())
```

Components

- `aiogram.fsm.scene.Scene` - represents a scene, contains handlers
- `aiogram.fsm.scene.SceneRegistry` - container for all scenes in the bot, used to register scenes and resolve them by name
- `aiogram.fsm.scene.ScenesManager` - manages scenes for each user, used to enter, leave and resolve current scene for user
- `aiogram.fsm.scene.SceneConfig` - scene configuration, used to configure scene
- `aiogram.fsm.scene.SceneWizard` - scene wizard, used to interact with user in scene from active scene handler
- Markers - marker for scene handlers, used to mark scene handlers

`class aiogram.fsm.scene.`**`Scene`**(*wizard:* SceneWizard)

Represents a scene in a conversation flow.

A scene is a specific state in a conversation where certain actions can take place.

Each scene has a set of filters that determine when it should be triggered, and a set of handlers that define the actions to be executed when the scene is active.

Note

This class is not meant to be used directly. Instead, it should be subclassed to define custom scenes.

`classmethod` **`add_to_router`**(*router:* Router) → None

Adds the scene to the given router.

Parameters
`router`

Returns

`classmethod` **`as_handler`**(***kwargs: Any*) → Callable[[...], Any]

Create an entry point handler for the scene, can be used to simplify the handler that starts the scene.

```
>>> router.message.register(MyScene.as_handler(), Command("start"))
```

`classmethod` **`as_router`**(*name: str | None = None*) → *Router*

Returns the scene as a router.

Returns
new router

`class aiogram.fsm.scene.`**`SceneRegistry`**(*router:* Router, *register_on_add: bool = True*)

A class that represents a registry for scenes in a Telegram bot.

`add`(**scenes: Type[*Scene*]*, *router:* Router | *None = None*) → None

This method adds the specified scenes to the registry and optionally registers it to the router.

If a scene with the same state already exists in the registry, a SceneException is raised.

> **Warning**
>
> If the router is not specified, the scenes will not be registered to the router. You will need to include the scenes manually to the router or use the register method.

Parameters

- **scenes** – A variable length parameter that accepts one or more types of scenes. These scenes are instances of the Scene class.
- **router** – An optional parameter that specifies the router to which the scenes should be added.

Returns
None

`get`(*scene: Type[Scene] | str | None*) → Type[*Scene*]

This method returns the registered Scene object for the specified scene. The scene parameter can be either a Scene object or a string representing the name of the scene. If a Scene object is provided, the state attribute of the SceneConfig object associated with the Scene object will be used as the scene name. If None or an invalid type is provided, a SceneException will be raised.

If the specified scene is not registered in the SceneRegistry object, a SceneException will be raised.

Parameters
scene – A Scene object or a string representing the name of the scene.

Returns
The registered Scene object corresponding to the given scene parameter.

`register`(**scenes: Type[Scene]*) → None

Registers one or more scenes to the SceneRegistry.

Parameters
scenes – One or more scene classes to register.

Returns
None

`class aiogram.fsm.scene.ScenesManager`(*registry: SceneRegistry, update_type: str, event: TelegramObject, state: FSMContext, data: Dict[str, Any]*)

The ScenesManager class is responsible for managing scenes in an application. It provides methods for entering and exiting scenes, as well as retrieving the active scene.

`async close`(***kwargs: Any*) → None

Close method is used to exit the currently active scene in the ScenesManager.

Parameters
kwargs – Additional keyword arguments passed to the scene's exit method.

Returns
None

`async enter`(*scene_type: Type[Scene] | str | None, _check_active: bool = True, **kwargs: Any*) → None

Enters the specified scene.

Parameters

- **scene_type** – Optional Type[Scene] or str representing the scene type to enter.

- **`_check_active`** – Optional bool indicating whether to check if there is an active scene to exit before entering the new scene. Defaults to True.
- **`kwargs`** – Additional keyword arguments to pass to the scene's wizard.enter() method.

Returns
None

`class aiogram.fsm.scene.`**`SceneConfig`**(*state: 'Optional[str]', handlers: 'List[HandlerContainer]', actions: 'Dict[SceneAction, Dict[str, CallableObject]]', reset_data_on_enter: 'Optional[bool]' = None, reset_history_on_enter: 'Optional[bool]' = None, callback_query_without_state: 'Optional[bool]' = None*)

`actions: Dict[SceneAction, Dict[str, CallableObject]]`
Scene actions

`callback_query_without_state: bool | None = None`
Allow callback query without state

`handlers: List[HandlerContainer]`
Scene handlers

`reset_data_on_enter: bool | None = None`
Reset scene data on enter

`reset_history_on_enter: bool | None = None`
Reset scene history on enter

`state: str | None`
Scene state

`class aiogram.fsm.scene.`**`SceneWizard`**(*scene_config: SceneConfig, manager: ScenesManager, state: FSMContext, update_type: str, event: TelegramObject, data: Dict[str, Any]*)

A class that represents a wizard for managing scenes in a Telegram bot.

Instance of this class is passed to each scene as a parameter. So, you can use it to transition between scenes, get and set data, etc.

> **Note**
>
> This class is not meant to be used directly. Instead, it should be used as a parameter in the scene constructor.

`async` **`back`**(***kwargs: Any*) → None
This method is used to go back to the previous scene.

Parameters
`kwargs` – Keyword arguments that can be passed to the method.

Returns
None

`async` **`clear_data`**() → None
Clears the data.

Returns
None

`async enter(**kwargs: Any) → None`

Enter method is used to transition into a scene in the SceneWizard class. It sets the state, clears data and history if specified, and triggers entering event of the scene.

Parameters
`kwargs` – Additional keyword arguments.

Returns
None

`async exit(**kwargs: Any) → None`

Exit the current scene and enter the default scene/state.

Parameters
`kwargs` – Additional keyword arguments.

Returns
None

`async get_data() → Dict[str, Any]`

This method returns the data stored in the current state.

Returns
A dictionary containing the data stored in the scene state.

`async goto(scene: Type[Scene] | str, **kwargs: Any) → None`

The *goto* method transitions to a new scene. It first calls the *leave* method to perform any necessary cleanup in the current scene, then calls the *enter* event to enter the specified scene.

Parameters

- `scene` – The scene to transition to. Can be either a *Scene* instance or a string representing the scene.
- `kwargs` – Additional keyword arguments to pass to the *enter* method of the scene manager.

Returns
None

`async leave(_with_history: bool = True, **kwargs: Any) → None`

Leaves the current scene. This method is used to exit a scene and transition to the next scene.

Parameters

- `_with_history` – Whether to include history in the snapshot. Defaults to True.
- `kwargs` – Additional keyword arguments.

Returns
None

`async retake(**kwargs: Any) → None`

This method allows to re-enter the current scene.

Parameters
`kwargs` – Additional keyword arguments to pass to the scene.

Returns
None

`async set_data(data: Dict[str, Any]) → None`

Sets custom data in the current state.

Parameters
`data` – A dictionary containing the custom data to be set in the current state.

Returns
None

`async update_data`(*data: Dict[str, Any] | None = None*, ***kwargs: Any*) → Dict[str, Any]

This method updates the data stored in the current state

Parameters

- **`data`** – Optional dictionary of data to update.
- **`kwargs`** – Additional key-value pairs of data to update.

Returns
Dictionary of updated data

Markers

Markers are similar to the Router event registering mechanism, but they are used to mark scene handlers in the Scene class.

It can be imported from `from aiogram.fsm.scene import on` and should be used as decorator.

Allowed event types:

- message
- edited_message
- channel_post
- edited_channel_post
- inline_query
- chosen_inline_result
- callback_query
- shipping_query
- pre_checkout_query
- poll
- poll_answer
- my_chat_member
- chat_member
- chat_join_request

Each event type can be filtered in the same way as in the Router.

Also each event type can be marked as scene entry point, exit point or leave point.

If you want to mark the scene can be entered from message or inline query, you should use `on.message` or `on.inline_query` marker:

```
class MyScene(Scene, name="my_scene"):
    @on.message.enter()
    async def on_enter(self, message: types.Message):
        pass

    @on.callback_query.enter()
    async def on_enter(self, callback_query: types.CallbackQuery):
        pass
```

Scene has only three points for transitions:

- enter point - when user enters to the scene
- leave point - when user leaves the scene and the enter another scene
- exit point - when user exits from the scene

2.4.7 Middlewares

aiogram provides powerful mechanism for customizing event handlers via middlewares.

Middlewares in bot framework seems like Middlewares mechanism in web-frameworks like aiohttp, fastapi, Django or etc.) with small difference - here is implemented two layers of middlewares (before and after filters).

Note

Middleware is function that triggered on every event received from Telegram Bot API in many points on processing pipeline.

Base theory

As many books and other literature in internet says:

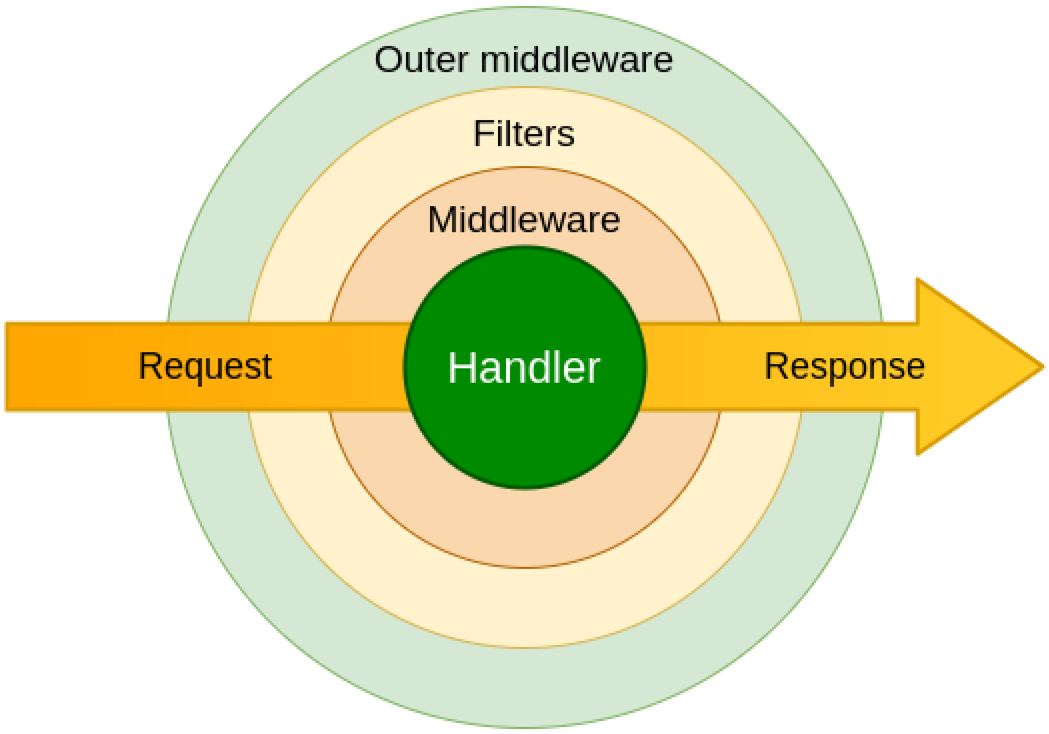
> Middleware is reusable software that leverages patterns and frameworks to bridge the gap between the functional requirements of applications and the underlying operating systems, network protocol stacks, and databases.

Middleware can modify, extend or reject processing event in many places of pipeline.

Basics

Middleware instance can be applied for every type of Telegram Event (Update, Message, etc.) in two places

1. Outer scope - before processing filters (`<router>.<event>.outer_middleware(...)`)
2. Inner scope - after processing filters but before handler (`<router>.<event>.middleware(...)`)



Attention

Middleware should be subclass of `BaseMiddleware` (`from aiogram import BaseMiddleware`) or any async callable

Arguments specification

class `aiogram.dispatcher.middlewares.base.`**`BaseMiddleware`**

Bases: `ABC`

Generic middleware class

abstract async **`__call__`**(*handler: Callable[[TelegramObject, Dict[str, Any]], Awaitable[Any]], event: TelegramObject, data: Dict[str, Any]*) → Any

Execute middleware

Parameters

- **handler** – Wrapped handler in middlewares chain
- **event** – Incoming event (Subclass of `aiogram.types.base.TelegramObject`)
- **data** – Contextual data. Will be mapped to handler arguments

Returns

`Any`

Examples

Danger

Middleware should always call `await handler(event, data)` to propagate event for next middleware/handler. If you want to stop processing event in middleware you should not call `await handler(event, data)`.

Class-based

```
from aiogram import BaseMiddleware
from aiogram.types import Message


class CounterMiddleware(BaseMiddleware):
    def __init__(self) -> None:
        self.counter = 0

    async def __call__(
        self,
        handler: Callable[[Message, Dict[str, Any]], Awaitable[Any]],
        event: Message,
        data: Dict[str, Any]
    ) -> Any:
        self.counter += 1
        data['counter'] = self.counter
        return await handler(event, data)
```

and then

```
router = Router()
router.message.middleware(CounterMiddleware())
```

Function-based

```
@dispatcher.update.outer_middleware()
async def database_transaction_middleware(
    handler: Callable[[Update, Dict[str, Any]], Awaitable[Any]],
    event: Update,
    data: Dict[str, Any]
) -> Any:
    async with database.transaction():
        return await handler(event, data)
```

Facts

1. Middlewares from outer scope will be called on every incoming event
2. Middlewares from inner scope will be called only when filters pass
3. Inner middlewares is always calls for *aiogram.types.update.Update* event type in due to all incoming updates going to specific event type handler through built in update handler

2.4.8 Errors

Handling errors

Is recommended way that you should use errors inside handlers using try-except block, but in common cases you can use global errors handler at router or dispatcher level.

If you specify errors handler for router - it will be used for all handlers inside this router.

If you specify errors handler for dispatcher - it will be used for all handlers inside all routers.

```
@router.error(ExceptionTypeFilter(MyCustomException), F.update.message.as_("message"))
async def handle_my_custom_exception(event: ErrorEvent, message: Message):
    # do something with error
    await message.answer("Oops, something went wrong!")


@router.error()
async def error_handler(event: ErrorEvent):
    logger.critical("Critical error caused by %s", event.exception, exc_info=True)
    # do something with error
    ...
```

ErrorEvent

class aiogram.types.error_event.**ErrorEvent**(**, update: Update, exception: Exception, **extra_data: Any*)

Internal event, should be used to receive errors while processing Updates from Telegram

Source: https://core.telegram.org/bots/api#error-event

update: *Update*

Received update

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

exception: Exception

Exception

Error types

exception `aiogram.exceptions.`**`AiogramError`**

Base exception for all aiogram errors.

exception `aiogram.exceptions.`**`DetailedAiogramError`**(*message: str*)

Base exception for all aiogram errors with detailed message.

exception `aiogram.exceptions.`**`CallbackAnswerException`**

Exception for callback answer.

exception `aiogram.exceptions.`**`SceneException`**

Exception for scenes.

exception `aiogram.exceptions.`**`UnsupportedKeywordArgument`**(*message: str*)

Exception raised when a keyword argument is passed as filter.

exception `aiogram.exceptions.`**`TelegramAPIError`**(*method: TelegramMethod*, *message: str*)

Base exception for all Telegram API errors.

exception `aiogram.exceptions.`**`TelegramNetworkError`**(*method: TelegramMethod*, *message: str*)

Base exception for all Telegram network errors.

exception `aiogram.exceptions.`**`TelegramRetryAfter`**(*method: TelegramMethod*, *message: str*, *retry_after: int*)

Exception raised when flood control exceeds.

exception `aiogram.exceptions.`**`TelegramMigrateToChat`**(*method: TelegramMethod*, *message: str*, *migrate_to_chat_id: int*)

Exception raised when chat has been migrated to a supergroup.

exception `aiogram.exceptions.`**`TelegramBadRequest`**(*method: TelegramMethod*, *message: str*)

Exception raised when request is malformed.

exception `aiogram.exceptions.`**`TelegramNotFound`**(*method: TelegramMethod*, *message: str*)

Exception raised when chat, message, user, etc. not found.

exception `aiogram.exceptions.`**`TelegramConflictError`**(*method: TelegramMethod*, *message: str*)

Exception raised when bot token is already used by another application in polling mode.

exception `aiogram.exceptions.`**`TelegramUnauthorizedError`**(*method: TelegramMethod*, *message: str*)

Exception raised when bot token is invalid.

exception `aiogram.exceptions.`**`TelegramForbiddenError`**(*method: TelegramMethod*, *message: str*)

Exception raised when bot is kicked from chat or etc.

exception `aiogram.exceptions.`**`TelegramServerError`**(*method: TelegramMethod*, *message: str*)

Exception raised when Telegram server returns 5xx error.

exception `aiogram.exceptions.`**`RestartingTelegram`**(*method: TelegramMethod*, *message: str*)

Exception raised when Telegram server is restarting.

It seems like this error is not used by Telegram anymore, but it’s still here for backward compatibility.

Currently, you should expect that Telegram can raise RetryAfter (with timeout 5 seconds)
error instead of this one.

exception aiogram.exceptions.**TelegramEntityTooLarge**(*method: TelegramMethod*, *message: str*)

Exception raised when you are trying to send a file that is too large.

exception aiogram.exceptions.**ClientDecodeError**(*message: str*, *original: Exception*, *data: Any*)

Exception raised when client can't decode response. (Malformed response, etc.)

2.4.9 Flags

Flags is a markers for handlers that can be used in *middlewares* or special *utilities* to make classification of the handlers.

Flags can be added to the handler via *decorators*, *handlers registration* or filters.

Via decorators

For example mark handler with *chat_action* flag

```
from aiogram import flags

@flags.chat_action
async def my_handler(message: Message)
```

Or just for rate-limit or something else

```
from aiogram import flags

@flags.rate_limit(rate=2, key="something")
async def my_handler(message: Message)
```

Via handler registration method

```
@router.message(..., flags={'chat_action': 'typing', 'rate_limit': {'rate': 5}})
```

Via filters

```
class Command(Filter):
    ...

    def update_handler_flags(self, flags: Dict[str, Any]) -> None:
        commands = flags.setdefault("commands", [])
        commands.append(self)
```

Use in middlewares

`aiogram.dispatcher.flags.`**`check_flags`**(*handler: HandlerObject* | *Dict[str, Any]*, *magic: MagicFilter*) → Any

Check flags via magic filter

Parameters

- **`handler`** – handler object or data
- **`magic`** – instance of the magic

Returns
the result of magic filter check

`aiogram.dispatcher.flags.`**`extract_flags`**(*handler: HandlerObject* | *Dict[str, Any]*) → Dict[str, Any]

Extract flags from handler or middleware context data

Parameters
`handler` – handler object or data

Returns
dictionary with all handler flags

`aiogram.dispatcher.flags.`**`get_flag`**(*handler: HandlerObject* | *Dict[str, Any]*, *name: str*, ***, *default: Any* | *None = None*) → Any

Get flag by name

Parameters

- **`handler`** – handler object or data
- **`name`** – name of the flag
- **`default`** – default value (None)

Returns
value of the flag or default

Example in middlewares

```python
async def my_middleware(handler, event, data):
    typing = get_flag(data, "typing")  # Check that handler marked with `typing` flag
    if not typing:
        return await handler(event, data)

    async with ChatActionSender.typing(chat_id=event.chat.id):
        return await handler(event, data)
```

Use in utilities

For example you can collect all registered commands with handler description and then it can be used for generating commands help

```
def collect_commands(router: Router) -> Generator[Tuple[Command, str], None, None]:
    for handler in router.message.handlers:
        if "commands" not in handler.flags:  # ignore all handler without commands
            continue
        # the Command filter adds the flag with list of commands attached to the handler
        for command in handler.flags["commands"]:
            yield command, handler.callback.__doc__ or ""
    # Recursively extract commands from nested routers
    for sub_router in router.sub_routers:
        yield from collect_commands(sub_router)
```

2.4.10 Webhook

Telegram Bot API supports webhook. If you set webhook for your bot, Telegram will send updates to the specified url. You can use *aiogram.methods.set_webhook.SetWebhook()* method to specify a url and receive incoming updates on it.

Note

If you use webhook, you can't use long polling at the same time.

Before start i'll recommend you to read official Telegram's documentation about webhook

After you read it, you can start to read this section.

Generally to use webhook with aiogram you should use any async web framework. By out of the box aiogram has an aiohttp integration, so we'll use it.

Note

You can use any async web framework you want, but you should write your own integration if you don't use aiohttp.

aiohttp integration

Out of the box aiogram has aiohttp integration, so you can use it.

Here is available few ways to do it using different implementations of the webhook controller:

- *aiogram.webhook.aiohttp_server.BaseRequestHandler* - Abstract class for aiohttp webhook controller
- *aiogram.webhook.aiohttp_server.SimpleRequestHandler* - Simple webhook controller, uses single Bot instance
- *aiogram.webhook.aiohttp_server.TokenBasedRequestHandler* - Token based webhook controller, uses multiple Bot instances and tokens

You can use it as is or inherit from it and override some methods.

class aiogram.webhook.aiohttp_server.**BaseRequestHandler**(*dispatcher: Dispatcher*, *handle_in_background: bool = False*, ***data: Any*)

__init__(*dispatcher: Dispatcher*, *handle_in_background: bool = False*, ***data: Any*) → None

Base handler that helps to handle incoming request from aiohttp and propagate it to the Dispatcher

Parameters

- **dispatcher** – instance of `aiogram.dispatcher.dispatcher.Dispatcher`
- **handle_in_background** – immediately responds to the Telegram instead of a waiting end of a handler process

register(*app: None*, */*, *path: str*, ***kwargs: Any*) → None

Register route and shutdown callback

Parameters

- **app** – instance of aiohttp Application
- **path** – route path
- **kwargs**

abstract async resolve_bot(*request: Request*) → Bot

This method should be implemented in subclasses of this class.

Resolve Bot instance from request.

Parameters
request

Returns
Bot instance

class aiogram.webhook.aiohttp_server.**SimpleRequestHandler**(*dispatcher: Dispatcher*, *bot: Bot*, *handle_in_background: bool = True*, *secret_token: str | None = None*, ***data: Any*)

__init__(*dispatcher: Dispatcher*, *bot: Bot*, *handle_in_background: bool = True*, *secret_token: str | None = None*, ***data: Any*) → None

Handler for single Bot instance

Parameters

- **dispatcher** – instance of `aiogram.dispatcher.dispatcher.Dispatcher`
- **handle_in_background** – immediately responds to the Telegram instead of a waiting end of handler process
- **bot** – instance of `aiogram.client.bot.Bot`

async close() → None

Close bot session

register(*app: None*, */*, *path: str*, ***kwargs: Any*) → None

Register route and shutdown callback

Parameters

- **app** – instance of aiohttp Application

- **path** – route path
- **kwargs**

async resolve_bot(*request: Request*) → Bot

This method should be implemented in subclasses of this class.

Resolve Bot instance from request.

Parameters
request

Returns
Bot instance

class aiogram.webhook.aiohttp_server.**TokenBasedRequestHandler**(*dispatcher: Dispatcher*, *handle_in_background: bool = True*, *bot_settings: Dict[str, Any] | None = None*, ***data: Any*)

__init__(*dispatcher: Dispatcher*, *handle_in_background: bool = True*, *bot_settings: Dict[str, Any] | None = None*, ***data: Any*) → None

Handler that supports multiple bots the context will be resolved from path variable 'bot_token'

Note

This handler is not recommended in due to token is available in URL and can be logged by reverse proxy server or other middleware.

Parameters

- **dispatcher** – instance of *aiogram.dispatcher.dispatcher.Dispatcher*
- **handle_in_background** – immediately responds to the Telegram instead of a waiting end of handler process
- **bot_settings** – kwargs that will be passed to new Bot instance

register(*app: None*, */*, *path: str*, ***kwargs: Any*) → None

Validate path, register route and shutdown callback

Parameters

- **app** – instance of aiohttp Application
- **path** – route path
- **kwargs**

async resolve_bot(*request: Request*) → Bot

Get bot token from a path and create or get from cache Bot instance

Parameters
request

Returns

Security

Telegram supports two methods to verify incoming requests that they are from Telegram:

Using a secret token

When you set webhook, you can specify a secret token and then use it to verify incoming requests.

Using IP filtering

You can specify a list of IP addresses from which you expect incoming requests, and then use it to verify incoming requests.

It can be acy using firewall rules or nginx configuration or middleware on application level.

So, aiogram has an implementation of the IP filtering middleware for aiohttp.

`aiogram.webhook.aiohttp_server.`**`ip_filter_middleware`**(*ip_filter: IPFilter*) → Callable[[Request, Callable[[Request], Awaitable[StreamResponse]]], Awaitable[Any]]

Parameters
`ip_filter`

Returns

`class` `aiogram.webhook.security.`**`IPFilter`**(*ips: Sequence[str | IPv4Network | IPv4Address] | None = None*)

`__init__`(*ips: Sequence[str | IPv4Network | IPv4Address] | None = None*)

Examples

Behind reverse proxy

In this example we'll use aiohttp as web framework and nginx as reverse proxy.

```
"""
This example shows how to use webhook on behind of any reverse proxy (nginx, traefik,␣
↪ingress etc.)
"""

import logging
import sys
from os import getenv

from aiohttp import web

from aiogram import Bot, Dispatcher, Router
from aiogram.client.default import DefaultBotProperties
from aiogram.enums import ParseMode
from aiogram.filters import CommandStart
from aiogram.types import Message
```

(continues on next page)

(continued from previous page)

```
from aiogram.utils.markdown import hbold
from aiogram.webhook.aiohttp_server import SimpleRequestHandler, setup_application

# Bot token can be obtained via https://t.me/BotFather
TOKEN = getenv("BOT_TOKEN")

# Webserver settings
# bind localhost only to prevent any external access
WEB_SERVER_HOST = "127.0.0.1"
# Port for incoming request from reverse proxy. Should be any available port
WEB_SERVER_PORT = 8080

# Path to webhook route, on which Telegram will send requests
WEBHOOK_PATH = "/webhook"
# Secret key to validate requests from Telegram (optional)
WEBHOOK_SECRET = "my-secret"
# Base URL for webhook will be used to generate webhook URL for Telegram,
# in this example it is used public DNS with HTTPS support
BASE_WEBHOOK_URL = "https://aiogram.dev"

# All handlers should be attached to the Router (or Dispatcher)
router = Router()


@router.message(CommandStart())
async def command_start_handler(message: Message) -> None:
    """
    This handler receives messages with `/start` command
    """
    # Most event objects have aliases for API methods that can be called in events'␣
↪context
    # For example if you want to answer to incoming message you can use `message.answer(.
↪..)` alias
    # and the target chat will be passed to :ref:`aiogram.methods.send_message.
↪SendMessage`
    # method automatically or call API method directly via
    # Bot instance: `bot.send_message(chat_id=message.chat.id, ...)`
    await message.answer(f"Hello, {hbold(message.from_user.full_name)}!")


@router.message()
async def echo_handler(message: Message) -> None:
    """
    Handler will forward receive a message back to the sender

    By default, message handler will handle all message types (like text, photo, sticker␣
↪etc.)
    """
    try:
        # Send a copy of the received message
        await message.send_copy(chat_id=message.chat.id)
    except TypeError:
```

(continues on next page)

(continued from previous page)

```
        # But not all the types is supported to be copied so need to handle it
        await message.answer("Nice try!")


async def on_startup(bot: Bot) -> None:
    # If you have a self-signed SSL certificate, then you will need to send a public
    # certificate to Telegram
    await bot.set_webhook(f"{BASE_WEBHOOK_URL}{WEBHOOK_PATH}", secret_token=WEBHOOK_
↪SECRET)


def main() -> None:
    # Dispatcher is a root router
    dp = Dispatcher()
    # ... and all other routers should be attached to Dispatcher
    dp.include_router(router)

    # Register startup hook to initialize webhook
    dp.startup.register(on_startup)

    # Initialize Bot instance with default bot properties which will be passed to all␣
↪API calls
    bot = Bot(token=TOKEN, default=DefaultBotProperties(parse_mode=ParseMode.HTML))

    # Create aiohttp.web.Application instance
    app = web.Application()

    # Create an instance of request handler,
    # aiogram has few implementations for different cases of usage
    # In this example we use SimpleRequestHandler which is designed to handle simple␣
↪cases
    webhook_requests_handler = SimpleRequestHandler(
        dispatcher=dp,
        bot=bot,
        secret_token=WEBHOOK_SECRET,
    )
    # Register webhook handler on application
    webhook_requests_handler.register(app, path=WEBHOOK_PATH)

    # Mount dispatcher startup and shutdown hooks to aiohttp application
    setup_application(app, dp, bot=bot)

    # And finally start webserver
    web.run_app(app, host=WEB_SERVER_HOST, port=WEB_SERVER_PORT)


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, stream=sys.stdout)
    main()
```

When you use nginx as reverse proxy, you should set *proxy_pass* to your aiohttp server address.

```
location /webhook {
    proxy_set_header Host $http_host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_redirect off;
    proxy_buffering off;
    proxy_pass http://127.0.0.1:8080;
}
```

Without reverse proxy (not recommended)

In case without using reverse proxy, you can use aiohttp's ssl context.

Also this example contains usage with self-signed certificate.

```python
"""
This example shows how to use webhook with SSL certificate.
"""

import logging
import ssl
import sys
from os import getenv

from aiohttp import web

from aiogram import Bot, Dispatcher, Router
from aiogram.client.default import DefaultBotProperties
from aiogram.enums import ParseMode
from aiogram.filters import CommandStart
from aiogram.types import FSInputFile, Message
from aiogram.utils.markdown import hbold
from aiogram.webhook.aiohttp_server import SimpleRequestHandler, setup_application

# Bot token can be obtained via https://t.me/BotFather
TOKEN = getenv("BOT_TOKEN")

# Webserver settings
# bind localhost only to prevent any external access
WEB_SERVER_HOST = "127.0.0.1"
# Port for incoming request from reverse proxy. Should be any available port
WEB_SERVER_PORT = 8080

# Path to webhook route, on which Telegram will send requests
WEBHOOK_PATH = "/webhook"
# Secret key to validate requests from Telegram (optional)
WEBHOOK_SECRET = "my-secret"
# Base URL for webhook will be used to generate webhook URL for Telegram,
# in this example it is used public address with TLS support
BASE_WEBHOOK_URL = "https://aiogram.dev"

# Path to SSL certificate and private key for self-signed certificate.
WEBHOOK_SSL_CERT = "/path/to/cert.pem"
```

(continues on next page)

(continued from previous page)

```
WEBHOOK_SSL_PRIV = "/path/to/private.key"

# All handlers should be attached to the Router (or Dispatcher)
router = Router()


@router.message(CommandStart())
async def command_start_handler(message: Message) -> None:
    """
    This handler receives messages with `/start` command
    """
    # Most event objects have aliases for API methods that can be called in events'␣
↪context
    # For example if you want to answer to incoming message you can use `message.answer(.
↪..)` alias
    # and the target chat will be passed to :ref:`aiogram.methods.send_message.
↪SendMessage`
    # method automatically or call API method directly via
    # Bot instance: `bot.send_message(chat_id=message.chat.id, ...)`
    await message.answer(f"Hello, {hbold(message.from_user.full_name)}!")


@router.message()
async def echo_handler(message: Message) -> None:
    """
    Handler will forward receive a message back to the sender

    By default, message handler will handle all message types (like text, photo, sticker␣
↪etc.)
    """
    try:
        # Send a copy of the received message
        await message.send_copy(chat_id=message.chat.id)
    except TypeError:
        # But not all the types is supported to be copied so need to handle it
        await message.answer("Nice try!")


async def on_startup(bot: Bot) -> None:
    # In case when you have a self-signed SSL certificate, you need to send the␣
↪certificate
    # itself to Telegram servers for validation purposes
    # (see https://core.telegram.org/bots/self-signed)
    # But if you have a valid SSL certificate, you SHOULD NOT send it to Telegram␣
↪servers.
    await bot.set_webhook(
        f"{BASE_WEBHOOK_URL}{WEBHOOK_PATH}",
        certificate=FSInputFile(WEBHOOK_SSL_CERT),
        secret_token=WEBHOOK_SECRET,
    )
```

(continues on next page)

(continued from previous page)

```python
def main() -> None:
    # Dispatcher is a root router
    dp = Dispatcher()
    # ... and all other routers should be attached to Dispatcher
    dp.include_router(router)

    # Register startup hook to initialize webhook
    dp.startup.register(on_startup)

    # Initialize Bot instance with default bot properties which will be passed to all␣
↪API calls
    bot = Bot(token=TOKEN, default=DefaultBotProperties(parse_mode=ParseMode.HTML))

    # Create aiohttp.web.Application instance
    app = web.Application()

    # Create an instance of request handler,
    # aiogram has few implementations for different cases of usage
    # In this example we use SimpleRequestHandler which is designed to handle simple␣
↪cases
    webhook_requests_handler = SimpleRequestHandler(
        dispatcher=dp,
        bot=bot,
        secret_token=WEBHOOK_SECRET,
    )
    # Register webhook handler on application
    webhook_requests_handler.register(app, path=WEBHOOK_PATH)

    # Mount dispatcher startup and shutdown hooks to aiohttp application
    setup_application(app, dp, bot=bot)

    # Generate SSL context
    context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
    context.load_cert_chain(WEBHOOK_SSL_CERT, WEBHOOK_SSL_PRIV)

    # And finally start webserver
    web.run_app(app, host=WEB_SERVER_HOST, port=WEB_SERVER_PORT, ssl_context=context)


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, stream=sys.stdout)
    main()
```

With using other web framework

You can pass incoming request to aiogram's webhook controller from any web framework you want.

Read more about it in `aiogram.dispatcher.dispatcher.Dispatcher.feed_webhook_update()` or *`aiogram.dispatcher.dispatcher.Dispatcher.feed_update()`* methods.

```
update = Update.model_validate(await request.json(), context={"bot": bot})
await dispatcher.feed_update(update)
```

Note

If you want to use reply into webhook, you should check that result of the `feed_update` methods is an instance of API method and build `multipart/form-data` or `application/json` response body manually.

2.4.11 Class based handlers

A handler is a async callable which takes a event with contextual data and returns a response.

In **aiogram** it can be more than just an async function, these allow you to use classes which can be used as Telegram event handlers to structure your event handlers and reuse code by harnessing inheritance and mixins.

There are some base class based handlers what you need to use in your own handlers:

BaseHandler

Base handler is generic abstract class and should be used in all other class-based handlers.

Import: `from aiogram.handlers import BaseHandler`

By default you will need to override only method `async def handle(self) -> Any:  ...`

This class also has a default initializer and you don't need to change it. The initializer accepts the incoming event and all contextual data, which can be accessed from the handler through attributes: `event:  TelegramEvent` and `data: Dict[Any, str]`

If an instance of the bot is specified in context data or current context it can be accessed through *bot* class attribute.

Example

```
class MyHandler(BaseHandler[Message]):
    async def handle(self) -> Any:
         await self.event.answer("Hello!")
```

CallbackQueryHandler

class aiogram.handlers.callback_query.**CallbackQueryHandler**(*event: T*, ***kwargs: Any*)

There is base class for callback query handlers.

Example:

```
from aiogram.handlers import CallbackQueryHandler

...

@router.callback_query()
class MyHandler(CallbackQueryHandler):
    async def handle(self) -> Any: ...
```

property from_user: *User*

Is alias for *event.from_user*

property message: *MaybeInaccessibleMessage* **| None**

Is alias for *event.message*

property callback_data: str | None

Is alias for *event.data*

ChosenInlineResultHandler

There is base class for chosen inline result handlers.

Simple usage

```
from aiogram.handlers import ChosenInlineResultHandler

...

@router.chosen_inline_result()
class MyHandler(ChosenInlineResultHandler):
    async def handle(self) -> Any: ...
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.chat` is alias for `self.event.chat`
- `self.from_user` is alias for `self.event.from_user`

ErrorHandler

There is base class for error handlers.

Simple usage

```
from aiogram.handlers import ErrorHandler

...

@router.errors()
class MyHandler(ErrorHandler):
    async def handle(self) -> Any:
        log.exception(
            "Cause unexpected exception %s: %s",
            self.exception_name,
            self.exception_message
        )
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.exception_name` is alias for `self.event.__class__.__name__`
- `self.exception_message` is alias for `str(self.event)`

InlineQueryHandler

There is base class for inline query handlers.

Simple usage

```
from aiogram.handlers import InlineQueryHandler

...

@router.inline_query()
class MyHandler(InlineQueryHandler):
    async def handle(self) -> Any: ...
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.chat` is alias for `self.event.chat`
- `self.query` is alias for `self.event.query`

MessageHandler

There is base class for message handlers.

Simple usage

```
from aiogram.handlers import MessageHandler

...

@router.message()
class MyHandler(MessageHandler):
    async def handle(self) -> Any:
        return SendMessage(chat_id=self.chat.id, text="PASS")
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.chat` is alias for `self.event.chat`
- `self.from_user` is alias for `self.event.from_user`

PollHandler

There is base class for poll handlers.

Simple usage

```
from aiogram.handlers import PollHandler

...

@router.poll()
class MyHandler(PollHandler):
    async def handle(self) -> Any: ...
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.question` is alias for `self.event.question`
- `self.options` is alias for `self.event.options`

PreCheckoutQueryHandler

There is base class for callback query handlers.

Simple usage

```
from aiogram.handlers import PreCheckoutQueryHandler

...

@router.pre_checkout_query()
class MyHandler(PreCheckoutQueryHandler):
    async def handle(self) -> Any: ...
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.from_user` is alias for `self.event.from_user`

ShippingQueryHandler

There is base class for callback query handlers.

Simple usage

```
from aiogram.handlers import ShippingQueryHandler

...

@router.shipping_query()
class MyHandler(ShippingQueryHandler):
    async def handle(self) -> Any: ...
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.from_user` is alias for `self.event.from_user`

ChatMemberHandler

There is base class for chat member updated events.

Simple usage

```python
from aiogram.handlers import ChatMemberHandler

...

@router.chat_member()
@router.my_chat_member()
class MyHandler(ChatMemberHandler):
    async def handle(self) -> Any: ...
```

Extension

This base handler is subclass of *BaseHandler* with some extensions:

- `self.chat` is alias for `self.event.chat`

2.5 Utils

2.5.1 Keyboard builder

Keyboard builder helps to dynamically generate markup.

Note

Note that if you have static markup, it's best to define it explicitly rather than using builder, but if you have dynamic markup configuration, feel free to use builder as you wish.

Usage example

For example you want to generate inline keyboard with 10 buttons

```
builder = InlineKeyboardBuilder()

for index in range(1, 11):
    builder.button(text=f"Set {index}", callback_data=f"set:{index}")
```

then adjust this buttons to some grid, for example first line will have 3 buttons, the next lines will have 2 buttons

```
builder.adjust(3, 2)
```

also you can attach another builder to this one

```
another_builder = InlineKeyboardBuilder(...)...  # Another builder with some buttons
builder.attach(another_builder)
```

or you can attach some already generated markup

```
markup = InlineKeyboardMarkup(inline_keyboard=[...])  # Some markup
builder.attach(InlineKeyboardBuilder.from_markup(markup))
```

and finally you can export this markup to use it in your message

```
await message.answer("Some text here", reply_markup=builder.as_markup())
```

Reply keyboard builder has the same interface

Warning

Note that you can’t attach reply keyboard builder to inline keyboard builder and vice versa

Inline Keyboard

`class aiogram.utils.keyboard.`**`InlineKeyboardBuilder`**(*markup: List[List[*InlineKeyboardButton*]] | None = None*)

Inline keyboard builder inherits all methods from generic builder

`button`(*text: str*, *url: str | None = None*, *login_url:* LoginUrl *| None = None*, *callback_data: str |* CallbackData *| None = None*, *switch_inline_query: str | None = None*, *switch_inline_query_current_chat: str | None = None*, *callback_game:* CallbackGame *| None = None*, *pay: bool | None = None*, ***kwargs: Any*) → *aiogram.utils.keyboard.InlineKeyboardBuilder*

Add new inline button to markup

`as_markup`() → *aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup*

Construct an InlineKeyboardMarkup

`__init__`(*markup: List[List[*InlineKeyboardButton*]] | None = None*) → None

`add`(**buttons: ButtonType*) → KeyboardBuilder[ButtonType]

Add one or many buttons to markup.

Parameters
`buttons`

Returns

`adjust`(**sizes: int*, *repeat: bool = False*) → KeyboardBuilder[ButtonType]

Adjust previously added buttons to specific row sizes.

By default, when the sum of passed sizes is lower than buttons count the last one size will be used for tail of the markup. If repeat=True is passed - all sizes will be cycled when available more buttons count than all sizes

Parameters

- `sizes`
- `repeat`

Returns

`property buttons: Generator[ButtonType, None, None]`

Get flatten set of all buttons

Returns

`copy`() → *InlineKeyboardBuilder*

Make full copy of current builder with markup

Returns

`export`() → List[List[ButtonType]]

Export configured markup as list of lists of buttons

```
>>> builder = KeyboardBuilder(button_type=InlineKeyboardButton)
>>> ... # Add buttons to builder
>>> markup = InlineKeyboardMarkup(inline_keyboard=builder.export())
```

Returns

`classmethod from_markup`(*markup:* InlineKeyboardMarkup) → *InlineKeyboardBuilder*

Create builder from existing markup

Parameters
`markup`

Returns

`row`(**buttons: ButtonType*, *width: int | None = None*) → KeyboardBuilder[ButtonType]

Add row to markup

When too much buttons is passed it will be separated to many rows

Parameters

- `buttons`
- `width`

Returns

Reply Keyboard

class aiogram.utils.keyboard.**ReplyKeyboardBuilder**(*markup: List[List[*KeyboardButton*]] | None = None*)

Reply keyboard builder inherits all methods from generic builder

button(*text: str*, *request_contact: bool | None = None*, *request_location: bool | None = None*, *request_poll:* KeyboardButtonPollType *| None = None*, ***kwargs: Any*) → *aiogram.utils.keyboard.ReplyKeyboardBuilder*

Add new button to markup

as_markup() → *aiogram.types.reply_keyboard_markup.ReplyKeyboardMarkup*

Construct an ReplyKeyboardMarkup

__init__(*markup: List[List[*KeyboardButton*]] | None = None*) → None

add(**buttons: ButtonType*) → KeyboardBuilder[ButtonType]

Add one or many buttons to markup.

Parameters
buttons

Returns

adjust(**sizes: int*, *repeat: bool = False*) → KeyboardBuilder[ButtonType]

Adjust previously added buttons to specific row sizes.

By default, when the sum of passed sizes is lower than buttons count the last one size will be used for tail of the markup. If repeat=True is passed - all sizes will be cycled when available more buttons count than all sizes

Parameters

- **sizes**
- **repeat**

Returns

property buttons: Generator[ButtonType, None, None]

Get flatten set of all buttons

Returns

copy() → *ReplyKeyboardBuilder*

Make full copy of current builder with markup

Returns

export() → List[List[ButtonType]]

Export configured markup as list of lists of buttons

```
>>> builder = KeyboardBuilder(button_type=InlineKeyboardButton)
>>> ... # Add buttons to builder
>>> markup = InlineKeyboardMarkup(inline_keyboard=builder.export())
```

Returns

classmethod from_markup(*markup: ReplyKeyboardMarkup*) → *ReplyKeyboardBuilder*

Create builder from existing markup

Parameters
markup

Returns

row(**buttons: ButtonType, width: int | None = None*) → KeyboardBuilder[ButtonType]

Add row to markup

When too much buttons is passed it will be separated to many rows

Parameters

- **buttons**
- **width**

Returns

2.5.2 Translation

In order to make you bot translatable you have to add a minimal number of hooks to your Python code.

These hooks are called translation strings.

The aiogram translation utils is build on top of GNU gettext Python module and Babel library.

Installation

Babel is required to make simple way to extract translation strings from your code

Can be installed from pip directly:

```
pip install Babel
```

or as *aiogram* extra dependency:

```
pip install aiogram[i18n]
```

Make messages translatable

In order to gettext need to know what the strings should be translated you will need to write translation strings.

For example:

```python
from aiogram import html
from aiogram.utils.i18n import gettext as _

async def my_handler(message: Message) -> None:
    await message.answer(
        _("Hello, {name}!").format(
            name=html.quote(message.from_user.full_name)
        )
    )
```

Danger

f-strings can't be used as translations string because any dynamic variables should be added to message after getting translated message

Also if you want to use translated string in keyword- or magic- filters you will need to use lazy gettext calls:

```
from aiogram import F
from aiogram.utils.i18n import lazy_gettext as __

@router.message(F.text == __("My menu entry"))
...
```

Danger

Lazy gettext calls should always be used when the current language is not know at the moment

Danger

Lazy gettext can't be used as value for API methods or any Telegram Object (like *aiogram.types.inline_keyboard_button.InlineKeyboardButton* or etc.)

Working with plural forms

The *gettext* from *aiogram.utils.i18n* is the one alias for two functions _gettext_ and _ngettext_ of GNU gettext Python module. Therefore, the wrapper for message strings is the same *_()*. You need to pass three parameters to the function: a singular string, a plural string, and a value.

Configuring engine

After you messages is already done to use gettext your bot should know how to detect user language

On top of your application the instance of `aiogram.utils.i18n.I18n` should be created

```
i18n = I18n(path="locales", default_locale="en", domain="messages")
```

After that you will need to choose one of builtin I18n middleware or write your own.

Builtin middlewares:

SimpleI18nMiddleware

class aiogram.utils.i18n.middleware.**SimpleI18nMiddleware**(*i18n: I18n, i18n_key: str | None = 'i18n', middleware_key: str = 'i18n_middleware'*)

Simple I18n middleware.

Chooses language code from the User object received in event

`__init__`(*i18n: I18n*, *i18n_key: str | None = 'i18n'*, *middleware_key: str = 'i18n_middleware'*) → None

Create an instance of middleware

Parameters

- **`i18n`** – instance of I18n
- **`i18n_key`** – context key for I18n instance
- **`middleware_key`** – context key for this middleware

ConstI18nMiddleware

`class aiogram.utils.i18n.middleware.`**`ConstI18nMiddleware`**(*locale: str*, *i18n: I18n*, *i18n_key: str | None = 'i18n'*, *middleware_key: str = 'i18n_middleware'*)

Const middleware chooses statically defined locale

`__init__`(*locale: str*, *i18n: I18n*, *i18n_key: str | None = 'i18n'*, *middleware_key: str = 'i18n_middleware'*) → None

Create an instance of middleware

Parameters

- **`i18n`** – instance of I18n
- **`i18n_key`** – context key for I18n instance
- **`middleware_key`** – context key for this middleware

FSMI18nMiddleware

`class aiogram.utils.i18n.middleware.`**`FSMI18nMiddleware`**(*i18n: I18n*, *key: str = 'locale'*, *i18n_key: str | None = 'i18n'*, *middleware_key: str = 'i18n_middleware'*)

This middleware stores locale in the FSM storage

`__init__`(*i18n: I18n*, *key: str = 'locale'*, *i18n_key: str | None = 'i18n'*, *middleware_key: str = 'i18n_middleware'*) → None

Create an instance of middleware

Parameters

- **`i18n`** – instance of I18n
- **`i18n_key`** – context key for I18n instance
- **`middleware_key`** – context key for this middleware

`async` **`set_locale`**(*state: FSMContext*, *locale: str*) → None

Write new locale to the storage

Parameters

- **`state`** – instance of FSMContext
- **`locale`** – new locale

I18nMiddleware

or define you own based on abstract I18nMiddleware middleware:

class aiogram.utils.i18n.middleware.**I18nMiddleware**(*i18n: I18n*, *i18n_key: str | None = 'i18n'*, *middleware_key: str = 'i18n_middleware'*)

Abstract I18n middleware.

__init__(*i18n: I18n*, *i18n_key: str | None = 'i18n'*, *middleware_key: str = 'i18n_middleware'*) → None

Create an instance of middleware

Parameters

- **i18n** – instance of I18n
- **i18n_key** – context key for I18n instance
- **middleware_key** – context key for this middleware

abstract async get_locale(*event: TelegramObject*, *data: Dict[str, Any]*) → str

Detect current user locale based on event and context.

This method must be defined in child classes

Parameters

- **event**
- **data**

Returns

setup(*router: Router*, *exclude: Set[str] | None = None*) → *BaseMiddleware*

Register middleware for all events in the Router

Parameters

- **router**
- **exclude**

Returns

Deal with Babel

Step 1 Extract messages

```
pybabel extract --input-dirs=. -o locales/messages.pot
```

Here is `--input-dirs=.` - path to code and the `locales/messages.pot` is template where messages will be extracted and *messages* is translation domain.

Working with plural forms

Extracting with Pybabel all strings options:

- `-k _:1,1t -k _:1,2` - for both singular and plural
- `-k __` - for lazy strings

```
pybabel extract -k _:1,1t -k _:1,2 -k __ --input-dirs=. -o locales/messages.pot
```

Note

Some useful options:

- Add comments for translators, you can use another tag if you want (TR) `--add-comments=NOTE`
- Contact email for bugreport `--msgid-bugs-address=EMAIL`
- Disable comments with string location in code `--no-location`
- Copyrights `--copyright-holder=AUTHOR`
- Set project name `--project=MySuperBot`
- Set version `--version=2.2`

Step 2: Init language

```
pybabel init -i locales/messages.pot -d locales -D messages -l en
```

- `-i locales/messages.pot` - pre-generated template
- `-d locales` - translations directory
- `-D messages` - translations domain
- `-l en` - language. Can be changed to any other valid language code (For example `-l uk` for ukrainian language)

Step 3: Translate texts

To open .po file you can use basic text editor or any PO editor, e.g. Poedit

Just open the file named `locales/{language}/LC_MESSAGES/messages.po` and write translations

Step 4: Compile translations

```
pybabel compile -d locales -D messages
```

Step 5: Updating messages

When you change the code of your bot you need to update po & mo files

- Step 5.1: regenerate pot file: command from step 1
- **Step 5.2: update po files**

  ```
  pybabel update -d locales -D messages -i locales/messages.pot
  ```

- Step 5.3: update your translations: location and tools you know from step 3
- Step 5.4: compile mo files: command from step 4

2.5.3 Chat action sender

Sender

class `aiogram.utils.chat_action.`**`ChatActionSender`**(**, bot: Bot, chat_id: str | int, message_thread_id: int | None = None, action: str = 'typing', interval: float = 5.0, initial_sleep: float = 0.0*)

This utility helps to automatically send chat action until long actions is done to take acknowledge bot users the bot is doing something and not crashed.

Provides simply to use context manager.

Technically sender start background task with infinity loop which works until action will be finished and sends the chat action every 5 seconds.

`__init__`(**, bot: Bot, chat_id: str | int, message_thread_id: int | None = None, action: str = 'typing', interval: float = 5.0, initial_sleep: float = 0.0*) → None

Parameters

- **`bot`** – instance of the bot
- **`chat_id`** – target chat id
- **`message_thread_id`** – unique identifier for the target message thread; supergroups only
- **`action`** – chat action type
- **`interval`** – interval between iterations
- **`initial_sleep`** – sleep before first sending of the action

classmethod choose_sticker(*chat_id: int | str, bot: Bot, message_thread_id: int | None = None, interval: float = 5.0, initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *choose_sticker* action

classmethod find_location(*chat_id: int | str, bot: Bot, message_thread_id: int | None = None, interval: float = 5.0, initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *find_location* action

classmethod record_video(*chat_id: int | str, bot: Bot, message_thread_id: int | None = None, interval: float = 5.0, initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *record_video* action

classmethod record_video_note(*chat_id: int | str, bot: Bot, message_thread_id: int | None = None, interval: float = 5.0, initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *record_video_note* action

classmethod record_voice(*chat_id: int | str, bot: Bot, message_thread_id: int | None = None, interval: float = 5.0, initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *record_voice* action

classmethod typing(*chat_id: int | str, bot: Bot, message_thread_id: int | None = None, interval: float = 5.0, initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *typing* action

classmethod upload_document(*chat_id: int | str, bot: Bot, message_thread_id: int | None = None, interval: float = 5.0, initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *upload_document* action

classmethod upload_photo(*chat_id: int | str*, *bot: Bot*, *message_thread_id: int | None = None*, *interval: float = 5.0*, *initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *upload_photo* action

classmethod upload_video(*chat_id: int | str*, *bot: Bot*, *message_thread_id: int | None = None*, *interval: float = 5.0*, *initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *upload_video* action

classmethod upload_video_note(*chat_id: int | str*, *bot: Bot*, *message_thread_id: int | None = None*, *interval: float = 5.0*, *initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *upload_video_note* action

classmethod upload_voice(*chat_id: int | str*, *bot: Bot*, *message_thread_id: int | None = None*, *interval: float = 5.0*, *initial_sleep: float = 0.0*) → *ChatActionSender*

Create instance of the sender with *upload_voice* action

Usage

```
async with ChatActionSender.typing(bot=bot, chat_id=message.chat.id):
    # Do something...
    # Perform some long calculations
    await message.answer(result)
```

Middleware

class aiogram.utils.chat_action.**ChatActionMiddleware**

Helps to automatically use chat action sender for all message handlers

Usage

Before usa should be registered for the *message* event

```
<router or dispatcher>.message.middleware(ChatActionMiddleware())
```

After this action all handlers which works longer than *initial_sleep* will produce the '*typing*' chat action.

Also sender can be customized via flags feature for particular handler.

Change only action type:

```
@router.message(...)
@flags.chat_action("sticker")
async def my_handler(message: Message): ...
```

Change sender configuration:

```
@router.message(...)
@flags.chat_action(initial_sleep=2, action="upload_document", interval=3)
async def my_handler(message: Message): ...
```

2.5.4 WebApp

Telegram Bot API 6.0 announces a revolution in the development of chatbots using WebApp feature.

You can read more details on it in the official blog and documentation.

aiogram implements simple utils to remove headache with the data validation from Telegram WebApp on the backend side.

Usage

For example from frontend you will pass `application/x-www-form-urlencoded` POST request with `_auth` field in body and wants to return User info inside response as `application/json`

```
from aiogram.utils.web_app import safe_parse_webapp_init_data
from aiohttp.web_request import Request
from aiohttp.web_response import json_response

async def check_data_handler(request: Request):
    bot: Bot = request.app["bot"]

    data = await request.post()  # application/x-www-form-urlencoded
    try:
        data = safe_parse_webapp_init_data(token=bot.token, init_data=data["_auth"])
    except ValueError:
        return json_response({"ok": False, "err": "Unauthorized"}, status=401)
    return json_response({"ok": True, "data": data.user.dict()})
```

Functions

`aiogram.utils.web_app.`**`check_webapp_signature`**(*token: str*, *init_data: str*) → bool

Check incoming WebApp init data signature

Source: https://core.telegram.org/bots/webapps#validating-data-received-via-the-web-app

Parameters

- **`token`** – bot Token
- **`init_data`** – data from frontend to be validated

Returns

`aiogram.utils.web_app.`**`parse_webapp_init_data`**(*init_data: str, *, loads: ~typing.Callable[[...], ~typing.Any] = <function loads>*) → WebAppInitData

Parse WebApp init data and return it as WebAppInitData object

This method doesn't make any security check, so you shall not trust to this data, use `safe_parse_webapp_init_data` instead.

Parameters

- **`init_data`** – data from frontend to be parsed
- **`loads`**

Returns

aiogram.utils.web_app.**safe_parse_webapp_init_data**(*token: str, init_data: str, *, loads: ~typing.Callable[[...], ~typing.Any] = <function loads>*) → *WebAppInitData*

Validate raw WebApp init data and return it as WebAppInitData object

Raise `ValueError` when data is invalid

Parameters

- **token** – bot token
- **init_data** – data from frontend to be parsed and validated
- **loads**

Returns

Types

class aiogram.utils.web_app.**WebAppInitData**(***extra_data: Any*)

This object contains data that is transferred to the Web App when it is opened. It is empty if the Web App was launched from a keyboard button.

Source: https://core.telegram.org/bots/webapps#webappinitdata

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'defer_build': True, 'extra': 'allow', 'frozen': True, 'populate_by_name': True, 'use_enum_values': True, 'validate_assignment': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[Dict[str, FieldInfo]] = {'auth_date': FieldInfo(annotation=datetime, required=True), 'can_send_after': FieldInfo(annotation=Union[int, NoneType], required=False, default=None), 'chat': FieldInfo(annotation=Union[WebAppChat, NoneType], required=False, default=None), 'chat_instance': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'chat_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'hash': FieldInfo(annotation=str, required=True), 'query_id': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'receiver': FieldInfo(annotation=Union[WebAppUser, NoneType], required=False, default=None), 'start_param': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'user': FieldInfo(annotation=Union[WebAppUser, NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo] objects.

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*context: Any, /*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

query_id: str | None

A unique identifier for the Web App session, required for sending messages via the answerWebAppQuery method.

user: *WebAppUser* | None

An object containing data about the current user.

receiver: *WebAppUser* | None

An object containing data about the chat partner of the current user in the chat where the bot was launched via the attachment menu. Returned only for Web Apps launched via the attachment menu.

chat: *WebAppChat* | None

An object containing data about the chat where the bot was launched via the attachment menu. Returned for supergroups, channels, and group chats – only for Web Apps launched via the attachment menu.

chat_type: str | None

Type of the chat from which the Web App was opened. Can be either "sender" for a private chat with the user opening the link, "private", "group", "supergroup", or "channel". Returned only for Web Apps launched from direct links.

chat_instance: str | None

Global identifier, uniquely corresponding to the chat from which the Web App was opened. Returned only for Web Apps launched from a direct link.

start_param: str | None

The value of the startattach parameter, passed via link. Only returned for Web Apps when launched from the attachment menu via link. The value of the start_param parameter will also be passed in the GET-parameter tgWebAppStartParam, so the Web App can load the correct interface right away.

can_send_after: int | None

Time in seconds, after which a message can be sent via the answerWebAppQuery method.

auth_date: datetime

Unix time when the form was opened.

hash: str

A hash of all passed parameters, which the bot server can use to check their validity.

class aiogram.utils.web_app.**WebAppUser**(***extra_data: Any*)

This object contains the data of the Web App user.

Source: https://core.telegram.org/bots/webapps#webappuser

id: int

A unique identifier for the user or bot. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. It has at most 52 significant bits, so a 64-bit integer or a double-precision float type is safe for storing this identifier.

is_bot: bool | None

True, if this user is a bot. Returns in the receiver field only.

first_name: str

First name of the user or bot.

last_name: str | None

Last name of the user or bot.

username: str | None

Username of the user or bot.

language_code: str | None

IETF language tag of the user's language. Returns in user field only.

`is_premium: bool | None`

True, if this user is a Telegram Premium user.

`added_to_attachment_menu: bool | None`

True, if this user added the bot to the attachment menu.

`allows_write_to_pm: bool | None`

True, if this user allowed the bot to message them.

`model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

`model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'defer_build': True, 'extra': 'allow', 'frozen': True, 'populate_by_name': True, 'use_enum_values': True, 'validate_assignment': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`model_fields: ClassVar[Dict[str, FieldInfo]] = {'added_to_attachment_menu': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None), 'allows_write_to_pm': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None), 'first_name': FieldInfo(annotation=str, required=True), 'id': FieldInfo(annotation=int, required=True), 'is_bot': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None), 'is_premium': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None), 'language_code': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'last_name': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'photo_url': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'username': FieldInfo(annotation=Union[str, NoneType], required=False, default=None)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo] objects.

This replaces *Model.__fields__* from Pydantic V1.

`model_post_init`(*context: Any*, /) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

`photo_url: str | None`

URL of the user's profile photo. The photo can be in .jpeg or .svg formats. Only returned for Web Apps launched from the attachment menu.

`class aiogram.utils.web_app.WebAppChat`(***extra_data: Any*)

This object represents a chat.

Source: https://core.telegram.org/bots/webapps#webappchat

`id: int`

Unique identifier for this chat. This number may have more than 32 significant bits and some programming languages may have difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or double-precision float type are safe for storing this identifier.

`type: str`

Type of chat, can be either "group", "supergroup" or "channel"

title: str

Title of the chat

username: str | None

Username of the chat

photo_url: str | None

URL of the chat's photo. The photo can be in .jpeg or .svg formats. Only returned for Web Apps launched from the attachment menu.

model_computed_fields: ClassVar[Dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'defer_build': True, 'extra': 'allow', 'frozen': True, 'populate_by_name': True, 'use_enum_values': True, 'validate_assignment': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[Dict[str, FieldInfo]] = {'id': FieldInfo(annotation=int, required=True), 'photo_url': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'title': FieldInfo(annotation=str, required=True), 'type': FieldInfo(annotation=str, required=True), 'username': FieldInfo(annotation=Union[str, NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo] objects.

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*context: Any*, */*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

2.5.5 Callback answer

Helper for callback query handlers, can be useful in bots with a lot of callback handlers to automatically take answer to all requests.

Simple usage

For use, it is enough to register the inner middleware *aiogram.utils.callback_answer.CallbackAnswerMiddleware* in dispatcher or specific router:

```
dispatcher.callback_query.middleware(CallbackAnswerMiddleware())
```

After that all handled callback queries will be answered automatically after processing the handler.

Advanced usage

In some cases you need to have some non-standard response parameters, this can be done in several ways:

Global defaults

Change default parameters while initializing middleware, for example change answer to *pre* mode and text "OK":

```
dispatcher.callback_query.middleware(CallbackAnswerMiddleware(pre=True, text="OK"))
```

Look at `aiogram.utils.callback_answer.CallbackAnswerMiddleware` to get all available parameters

Handler specific

By using *flags* you can change the behavior for specific handler

```
@router.callback_query(<filters>)
@flags.callback_answer(text="Thanks", cache_time=30)
async def my_handler(query: CallbackQuery):
    ...
```

Flag arguments is the same as in `aiogram.utils.callback_answer.CallbackAnswerMiddleware` with additional one `disabled` to disable answer.

A special case

It is not always correct to answer the same in every case, so there is an option to change the answer inside the handler. You can get an instance of `aiogram.utils.callback_answer.CallbackAnswer` object inside handler and change whatever you want.

Danger

Note that is impossible to change callback answer attributes when you use `pre=True` mode.

```
@router.callback_query(<filters>)
async def my_handler(query: CallbackQuery, callback_answer: CallbackAnswer):
    ...
    if <everything is ok>:
        callback_answer.text = "All is ok"
    else:
        callback_answer.text = "Something wrong"
        callback_answer.cache_time = 10
```

Combine that all at once

For example you want to answer in most of cases before handler with text “” but at some cases need to answer after the handler with custom text, so you can do it:

```
dispatcher.callback_query.middleware(CallbackAnswerMiddleware(pre=True, text=""))

@router.callback_query(<filters>)
@flags.callback_answer(pre=False, cache_time=30)
async def my_handler(query: CallbackQuery):
    ...
    if <everything is ok>:
        callback_answer.text = "All is ok"
```

Description of objects

class aiogram.utils.callback_answer.**CallbackAnswerMiddleware**(*pre: bool = False*, *text: str | None = None*, *show_alert: bool | None = None*, *url: str | None = None*, *cache_time: int | None = None*)

Bases: `BaseMiddleware`

__init__(*pre: bool = False*, *text: str | None = None*, *show_alert: bool | None = None*, *url: str | None = None*, *cache_time: int | None = None*) → None

Inner middleware for callback query handlers, can be useful in bots with a lot of callback handlers to automatically take answer to all requests

Parameters

- **pre** – send answer before execute handler
- **text** – answer with text
- **show_alert** – show alert
- **url** – game url
- **cache_time** – cache answer for some time

class aiogram.utils.callback_answer.**CallbackAnswer**(*answered: bool*, *disabled: bool = False*, *text: str | None = None*, *show_alert: bool | None = None*, *url: str | None = None*, *cache_time: int | None = None*)

Bases: `object`

__init__(*answered: bool*, *disabled: bool = False*, *text: str | None = None*, *show_alert: bool | None = None*, *url: str | None = None*, *cache_time: int | None = None*) → None

Callback answer configuration

Parameters

- **answered** – this request is already answered by middleware
- **disabled** – answer will not be performed
- **text** – answer with text
- **show_alert** – show alert

- **url** – game url
- **cache_time** – cache answer for some time

disable() → None

Deactivate answering for this handler

property disabled: bool

Indicates that automatic answer is disabled in this handler

property answered: bool

Indicates that request is already answered by middleware

property text: str | None

Response text :return:

property show_alert: bool | None

Whether to display an alert

property url: str | None

Game url

property cache_time: int | None

Response cache time

2.5.6 Formatting

Make your message formatting flexible and simple

This instrument works on top of Message entities instead of using HTML or Markdown markups, you can easily construct your message and sent it to the Telegram without the need to remember tag parity (opening and closing) or escaping user input.

Usage

Basic scenario

Construct your message and send it to the Telegram.

```
content = Text("Hello, ", Bold(message.from_user.full_name), "!")
await message.answer(**content.as_kwargs())
```

Is the same as the next example, but without usage markup

```
await message.answer(
    text=f"Hello, <b>{html.quote(message.from_user.full_name)}!",
    parse_mode=ParseMode.HTML
)
```

Literally when you execute `as_kwargs` method the Text object is converted into text `Hello, Alex!` with entities list `[MessageEntity(type='bold', offset=7, length=4)]` and passed into dict which can be used as `**kwargs` in API call.

The complete list of elements is listed *on this page below*.

Advanced scenario

On top of base elements can be implemented content rendering structures, so, out of the box aiogram has a few already implemented functions that helps you to format your messages:

`aiogram.utils.formatting.`**`as_line`**`(`**items: Any*, *end: str = '\n'*, *sep: str = ''*`)` → *Text*

Wrap multiple nodes into line with `\n` at the end of line.

Parameters

- **`items`** – Text or Any
- **`end`** – ending of the line, by default is `\n`
- **`sep`** – separator between items, by default is empty string

Returns
Text

`aiogram.utils.formatting.`**`as_list`**`(`**items: Any*, *sep: str = '\n'*`)` → *Text*

Wrap each element to separated lines

Parameters

- **`items`**
- **`sep`**

Returns

`aiogram.utils.formatting.`**`as_marked_list`**`(`**items: Any*, *marker: str = '- '*`)` → *Text*

Wrap elements as marked list

Parameters

- **`items`**
- **`marker`** – line marker, by default is '- '

Returns
Text

`aiogram.utils.formatting.`**`as_numbered_list`**`(`**items: Any*, *start: int = 1*, *fmt: str = '{}. '*`)` → *Text*

Wrap elements as numbered list

Parameters

- **`items`**
- **`start`** – initial number, by default 1
- **`fmt`** – number format, by default '{}. '

Returns
Text

`aiogram.utils.formatting.`**`as_section`**`(`*title: Any*, **body: Any*`)` → *Text*

Wrap elements as simple section, section has title and body

Parameters

- **`title`**
- **`body`**

Returns
Text

aiogram.utils.formatting.**as_marked_section**(*title: Any*, **body: Any*, *marker: str = '- '*) → *Text*

Wrap elements as section with marked list

Parameters

- **title**
- **body**
- **marker**

Returns

aiogram.utils.formatting.**as_numbered_section**(*title: Any*, **body: Any*, *start: int = 1*, *fmt: str = '{}. '*) → *Text*

Wrap elements as section with numbered list

Parameters

- **title**
- **body**
- **start**
- **fmt**

Returns

aiogram.utils.formatting.**as_key_value**(*key: Any*, *value: Any*) → *Text*

Wrap elements pair as key-value line. (<b>{key}:</b> {value})

Parameters

- **key**
- **value**

Returns
Text

and lets complete them all:

```
content = as_list(
    as_marked_section(
        Bold("Success:"),
        "Test 1",
        "Test 3",
        "Test 4",
        marker=" ",
    ),
    as_marked_section(
        Bold("Failed:"),
        "Test 2",
        marker=" ",
    ),
    as_marked_section(
        Bold("Summary:"),
        as_key_value("Total", 4),
```

(continues on next page)

(continued from previous page)

```
        as_key_value("Success", 3),
        as_key_value("Failed", 1),
        marker="  ",
    ),
    HashTag("#test"),
    sep="\n\n",
)
```

Will be rendered into:

Success:

 Test 1

 Test 3

 Test 4

Failed:

 Test 2

Summary:

Total: 4

Success: 3

Failed: 1

#test

Or as HTML:

```
<b>Success:</b>
 Test 1
 Test 3
 Test 4

<b>Failed:</b>
 Test 2

<b>Summary:</b>
  <b>Total:</b> 4
  <b>Success:</b> 3
  <b>Failed:</b> 1

#test
```

Available methods

class aiogram.utils.formatting.**Text**(**body: Any*, ***params: Any*)

Bases: `Iterable[Any]`

Simple text element

`__init__`(**body: Any*, ***params: Any*) → None

`render`(***, *_offset: int = 0*, *_sort: bool = True*, *_collect_entities: bool = True*) → Tuple[str, List[*MessageEntity*]]

Render elements tree as text with entities list

Returns

`as_kwargs`(***, *text_key: str = 'text'*, *entities_key: str = 'entities'*, *replace_parse_mode: bool = True*, *parse_mode_key: str = 'parse_mode'*) → Dict[str, Any]

Render elements tree as keyword arguments for usage in the API call, for example:

```
entities = Text(...)
await message.answer(**entities.as_kwargs())
```

Parameters

- **text_key**
- **entities_key**
- **replace_parse_mode**
- **parse_mode_key**

Returns

`as_html`() → str

Render elements tree as HTML markup

`as_markdown`() → str

Render elements tree as MarkdownV2 markup

Available elements

class aiogram.utils.formatting.**Text**(**body: Any*, ***params: Any*)

Bases: `Iterable[Any]`

Simple text element

class aiogram.utils.formatting.**HashTag**(**body: Any*, ***params: Any*)

Bases: *`Text`*

Hashtag element.

Warning

The value should always start with '#' symbol

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.HASHTAG*

class aiogram.utils.formatting.**CashTag**(**body: Any*, ***params: Any*)

Bases: *Text*

Cashtag element.

Warning

The value should always start with '$' symbol

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.CASHTAG*

class aiogram.utils.formatting.**BotCommand**(**body: Any*, ***params: Any*)

Bases: *Text*

Bot command element.

Warning

The value should always start with '/' symbol

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.BOT_COMMAND*

class aiogram.utils.formatting.**Url**(**body: Any*, ***params: Any*)

Bases: *Text*

Url element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.URL*

class aiogram.utils.formatting.**Email**(**body: Any*, ***params: Any*)

Bases: *Text*

Email element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.EMAIL*

class aiogram.utils.formatting.**PhoneNumber**(**body: Any*, ***params: Any*)

Bases: *Text*

Phone number element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.PHONE_NUMBER*

class aiogram.utils.formatting.**Bold**(**body: Any*, ***params: Any*)

Bases: *Text*

Bold element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.BOLD*

class aiogram.utils.formatting.**Italic**(**body: Any*, ***params: Any*)

Bases: *Text*

Italic element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.ITALIC*

class aiogram.utils.formatting.**Underline**(**body: Any*, ***params: Any*)

Bases: *Text*

Underline element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.UNDERLINE*

class aiogram.utils.formatting.**Strikethrough**(**body: Any*, ***params: Any*)

Bases: *Text*

Strikethrough element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.STRIKETHROUGH*

class aiogram.utils.formatting.**Spoiler**(**body: Any*, ***params: Any*)

Bases: *Text*

Spoiler element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.SPOILER*

class aiogram.utils.formatting.**Code**(**body: Any*, ***params: Any*)

Bases: *Text*

Code element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.CODE*

class aiogram.utils.formatting.**Pre**(**body: Any*, *language: str | None = None*, ***params: Any*)

Bases: *Text*

Pre element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.PRE*

class aiogram.utils.formatting.**TextLink**(**body: Any*, *url: str*, ***params: Any*)

Bases: *Text*

Text link element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.TEXT_LINK*

class aiogram.utils.formatting.**TextMention**(**body: Any*, *user: User*, ***params: Any*)

Bases: *Text*

Text mention element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.TEXT_MENTION*

class aiogram.utils.formatting.**CustomEmoji**(**body: Any*, *custom_emoji_id: str*, ***params: Any*)

Bases: *Text*

Custom emoji element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.CUSTOM_EMOJI*

class aiogram.utils.formatting.**BlockQuote**(**body: Any*, ***params: Any*)

Bases: *Text*

Block quote element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.BLOCKQUOTE*

class aiogram.utils.formatting.**ExpandableBlockQuote**(**body: Any*, ***params: Any*)

Bases: *Text*

Expandable block quote element.

Will be wrapped into *aiogram.types.message_entity.MessageEntity* with type *aiogram.enums.message_entity_type.MessageEntityType.EXPANDABLE_BLOCKQUOTE*

2.5.7 Media group builder

This module provides a builder for media groups, it can be used to build media groups for *aiogram.types.input_media_photo.InputMediaPhoto*, *aiogram.types.input_media_video.InputMediaVideo*, *aiogram.types.input_media_document.InputMediaDocument* and *aiogram.types.input_media_audio.InputMediaAudio*.

Warning

aiogram.types.input_media_animation.InputMediaAnimation is not supported yet in the Bot API to send as media group.

Usage

```
media_group = MediaGroupBuilder(caption="Media group caption")

# Add photo
media_group.add_photo(media="https://picsum.photos/200/300")
# Dynamically add photo with known type without using separate method
media_group.add(type="photo", media="https://picsum.photos/200/300")
# ... or video
media_group.add(type="video", media=FSInputFile("media/video.mp4"))
```

To send media group use *aiogram.methods.send_media_group.SendMediaGroup()* method, but when you use *aiogram.utils.media_group.MediaGroupBuilder* you should pass `media` argument as `media_group.build()`.

If you specify `caption` in *aiogram.utils.media_group.MediaGroupBuilder* it will be used as `caption` for first media in group.

```
await bot.send_media_group(chat_id=chat_id, media=media_group.build())
```

References

class aiogram.utils.media_group.**MediaGroupBuilder**(*media: List[*InputMediaAudio | InputMediaPhoto | InputMediaVideo | InputMediaDocument*] | None = None*, *caption: str | None = None*, *caption_entities: List[*MessageEntity*] | None = None*)

add(*, *type: Literal[InputMediaType.AUDIO]*, *media: str |* InputFile, *caption: str | None = None*, *parse_mode: str | None = UNSET_PARSE_MODE*, *caption_entities: List[*MessageEntity*] | None = None*, *duration: int | None = None*, *performer: str | None = None*, *title: str | None = None*, ***kwargs: Any*) → None

add(*, *type: Literal[InputMediaType.PHOTO]*, *media: str |* InputFile, *caption: str | None = None*, *parse_mode: str | None = UNSET_PARSE_MODE*, *caption_entities: List[*MessageEntity*] | None = None*, *has_spoiler: bool | None = None*, ***kwargs: Any*) → None

add(*, *type: Literal[InputMediaType.VIDEO]*, *media: str |* InputFile, *thumbnail:* InputFile *| str | None = None*, *caption: str | None = None*, *parse_mode: str | None = UNSET_PARSE_MODE*, *caption_entities: List[*MessageEntity*] | None = None*, *width: int | None = None*, *height: int | None = None*, *duration: int | None = None*, *supports_streaming: bool | None = None*, *has_spoiler: bool | None = None*, ***kwargs: Any*) → None

add(*, *type: Literal[InputMediaType.DOCUMENT]*, *media: str |* InputFile, *thumbnail:* InputFile *| str | None = None*, *caption: str | None = None*, *parse_mode: str | None = UNSET_PARSE_MODE*, *caption_entities: List[*MessageEntity*] | None = None*, *disable_content_type_detection: bool | None = None*, ***kwargs: Any*) → None

Add a media object to the media group.

Parameters

kwargs – Keyword arguments for the media object. The available keyword arguments depend on the media type.

Returns

None

add_audio(*media: str | ~aiogram.types.input_file.InputFile*, *thumbnail: ~aiogram.types.input_file.InputFile | None = None*, *caption: str | None = None*, *parse_mode: str | None = <Default('parse_mode')>*, *caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *duration: int | None = None*, *performer: str | None = None*, *title: str | None = None*, ***kwargs: ~typing.Any*) → None

Add an audio file to the media group.

Parameters

- **media** – File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name.

  More information on Sending Files »

- **thumbnail** – *Optional.* Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320.

- **caption** – *Optional*. Caption of the audio to be sent, 0-1024 characters after entities parsing
- **parse_mode** – *Optional*. Mode for parsing entities in the audio caption. See formatting options for more details.
- **caption_entities** – *Optional*. List of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **duration** – *Optional*. Duration of the audio in seconds
- **performer** – *Optional*. Performer of the audio
- **title** – *Optional*. Title of the audio

Returns
None

add_document(*media: str | ~aiogram.types.input_file.InputFile*, *thumbnail: ~aiogram.types.input_file.InputFile | None = None*, *caption: str | None = None*, *parse_mode: str | None = <Default('parse_mode')>*, *caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *disable_content_type_detection: bool | None = None*, ***kwargs: ~typing.Any*) → None

Add a document to the media group.

Parameters

- **media** – File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*
- **thumbnail** – *Optional*. Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – *Optional*. Caption of the document to be sent, 0-1024 characters after entities parsing
- **parse_mode** – *Optional*. Mode for parsing entities in the document caption. See formatting options for more details.
- **caption_entities** – *Optional*. List of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **disable_content_type_detection** – *Optional*. Disables automatic server-side content type detection for files uploaded using multipart/form-data. Always `True`, if the document is sent as part of an album.

Returns
None

add_photo(*media: str | ~aiogram.types.input_file.InputFile*, *caption: str | None = None*, *parse_mode: str | None = <Default('parse_mode')>*, *caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *has_spoiler: bool | None = None*, ***kwargs: ~typing.Any*) → None

Add a photo to the media group.

Parameters

- **media** – File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name.

  More information on Sending Files »

- **caption** – *Optional.* Caption of the photo to be sent, 0-1024 characters after entities parsing
- **parse_mode** – *Optional.* Mode for parsing entities in the photo caption. See formatting options for more details.
- **caption_entities** – *Optional.* List of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **has_spoiler** – *Optional.* Pass `True` if the photo needs to be covered with a spoiler animation

Returns
None

`add_video`(*media: str | ~aiogram.types.input_file.InputFile*, *thumbnail: ~aiogram.types.input_file.InputFile | None = None*, *caption: str | None = None*, *parse_mode: str | None = <Default('parse_mode')>*, *caption_entities: ~typing.List[~aiogram.types.message_entity.MessageEntity] | None = None*, *width: int | None = None*, *height: int | None = None*, *duration: int | None = None*, *supports_streaming: bool | None = None*, *has_spoiler: bool | None = None*, ***kwargs: ~typing.Any*) → None

Add a video to the media group.

Parameters

- **media** – File to send. Pass a file_id to send a file that exists on the Telegram servers (recommended), pass an HTTP URL for Telegram to get a file from the Internet, or pass 'attach://<file_attach_name>' to upload a new one using multipart/form-data under <file_attach_name> name. *More information on Sending Files »*
- **thumbnail** – *Optional.* Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass 'attach://<file_attach_name>' if the thumbnail was uploaded using multipart/form-data under <file_attach_name>. *More information on Sending Files »*
- **caption** – *Optional.* Caption of the video to be sent, 0-1024 characters after entities parsing
- **parse_mode** – *Optional.* Mode for parsing entities in the video caption. See formatting options for more details.
- **caption_entities** – *Optional.* List of special entities that appear in the caption, which can be specified instead of *parse_mode*
- **width** – *Optional.* Video width
- **height** – *Optional.* Video height
- **duration** – *Optional.* Video duration in seconds

- **supports_streaming** – *Optional*. Pass `True` if the uploaded video is suitable for streaming
- **has_spoiler** – *Optional*. Pass `True` if the video needs to be covered with a spoiler animation

Returns
None

build() → List[*InputMediaAudio* | *InputMediaPhoto* | *InputMediaVideo* | *InputMediaDocument*]

Builds a list of media objects for a media group.

Adds the caption to the first media object if it is present.

Returns
List of media objects.

2.5.8 Deep Linking

Telegram bots have a deep linking mechanism, that allows for passing additional parameters to the bot on startup. It could be a command that launches the bot — or an auth token to connect the user's Telegram account to their account on some external service.

You can read detailed description in the source: https://core.telegram.org/bots/features#deep-linking

We have added some utils to get deep links more handy.

Examples

Basic link example

```
from aiogram.utils.deep_linking import create_start_link

link = await create_start_link(bot, 'foo')

# result: 'https://t.me/MyBot?start=foo'
```

Encoded link

```
from aiogram.utils.deep_linking import create_start_link

link = await create_start_link(bot, 'foo', encode=True)
# result: 'https://t.me/MyBot?start=Zm9v'
```

Decode it back

```
from aiogram.utils.deep_linking import decode_payload
from aiogram.filters import CommandStart, CommandObject
from aiogram.types import Message

@router.message(CommandStart(deep_link=True))
async def handler(message: Message, command: CommandObject):
    args = command.args
    payload = decode_payload(args)
    await message.answer(f"Your payload: {payload}")
```

References

async aiogram.utils.deep_linking.**create_start_link**(*bot: Bot*, *payload: str*, *encode: bool = False*, *encoder: Callable[[bytes], bytes] | None = None*) → str

Create 'start' deep link with your payload.

If you need to encode payload or pass special characters -
set encode as True

Parameters

- **bot** – bot instance
- **payload** – args passed with /start
- **encode** – encode payload with base64url or custom encoder
- **encoder** – custom encoder callable

Returns
link

aiogram.utils.deep_linking.**decode_payload**(*payload: str*, *decoder: Callable[[bytes], bytes] | None = None*) → str

Decode URL-safe base64url payload with decoder.

2.5.9 Telegram object serialization

Serialization

To serialize Python object to Telegram object you can use pydantic serialization methods, for example:

```
message_data = { ... }  # Some message data as dict
message = Message.model_validate(message_data)
```

If you want to bind serialized object to the Bot instance, you can use context:

```
message_data = { ... }  # Some message data as dict
message = Message.model_validate(message_data, context={"bot": bot})
```

Deserialization

In cases when you need to deserialize Telegram object to Python object, you can use this module.

To convert Telegram object to Python object excluding files you can use `aiogram.utils.serialization.deserialize_telegram_object_to_python()` function.

`aiogram.utils.serialization.`**`deserialize_telegram_object_to_python`**(*obj: Any*, *default:* DefaultBotProperties | *None = None*, *include_api_method_name: bool = True*) → Any

Deserialize telegram object to JSON compatible Python object excluding files.

Parameters

- **`obj`** – The telegram object to be deserialized.
- **`default`** – Default bot properties should be passed only if you want to use custom defaults.
- **`include_api_method_name`** – Whether to include the API method name in the result.

Returns

The deserialized telegram object.

To convert Telegram object to Python object including files you can use `aiogram.utils.serialization.deserialize_telegram_object()` function, which returns `aiogram.utils.serialization.DeserializedTelegramObject` object.

`aiogram.utils.serialization.`**`deserialize_telegram_object`**(*obj: Any*, *default:* DefaultBotProperties | *None = None*, *include_api_method_name: bool = True*) → *DeserializedTelegramObject*

Deserialize Telegram Object to JSON compatible Python object.

Parameters

- **`obj`** – The object to be deserialized.
- **`default`** – Default bot properties should be passed only if you want to use custom defaults.
- **`include_api_method_name`** – Whether to include the API method name in the result.

Returns

The deserialized Telegram object.

`class` `aiogram.utils.serialization.`**`DeserializedTelegramObject`**(*data: Any*, *files: Dict[str,* InputFile*]*)

Represents a dumped Telegram object.

Parameters

- **`data`** (*`Any`*) – The dumped data of the Telegram object.
- **`files`** (*`Dict[str, InputFile]`*) – The dictionary containing the file names as keys and the corresponding *InputFile* objects as values.

2.6 Changelog

2.6.1 3.13.1 (2024-09-18)

Warning

Python 3.8 End of Life: Python 3.8 will reach its end of life (EOL) soon and will no longer be supported by aiogram in the next releases (1-2 months ETA).

Please upgrade to a newer version of Python to ensure compatibility and receive future updates.

Misc

- Increase max pydantic version support "<2.9" -> "<2.10" (only For Python >=3.9) #1576
- Bump aiofiles version upper bound to <24.2 #1577

Bugfixes

- Fixed *Default* object annotation resolution using *pydantic* #1579

2.6.2 3.13.0 (2024-09-08)

Features

-
  - Added updates about purchased paid media, represented by the class `aiogram.types.paid_media_purchased.PaidMediaPurchased` and the field `purchased_paid_media` in the class `aiogram.types.update.Update`.
  - Added the ability to specify a payload in `aiogram.methods.send_paid_media.SendPaidMedia` that is received back by the bot in `aiogram.types.transaction_partner_user.TransactionPartnerUser` and `purchased_paid_media` updates.
  - Added the field `prize_star_count` to the classes `aiogram.types.giveaway_created.GiveawayCreated`, `aiogram.types.giveaway.Giveaway`, `aiogram.types.giveaway_winners.GiveawayWinners` and `aiogram.types.chat_boost_source_giveaway.ChatBoostSourceGiveaway`.
  - Added the field `is_star_giveaway` to the class `aiogram.types.giveaway_completed.GiveawayCompleted`.

  #1510
- Added missing method aliases such as *.answer()*, *.reply()*, and others to *InaccessibleMessage*. This change ensures consistency and improves usability by aligning the functionality of *InaccessibleMessage* with the *Message* type. #1574

Bugfixes

- Fixed link preview options to use global defaults in various types and methods to use global defaults for *link_preview_options*. This change ensures consistency and enhances flexibility in handling link preview options across different components. #1543

2.6.3 3.12.0 (2024-08-16)

Features

- Added **message_thread_id** parameter to **message.get_url()**. #1451
- Added getting user from *chat_boost* with source *ChatBoostSourcePremium* in *UserContextMiddleware* for *EventContext* #1474
- Added full support of Bot API 7.8
  - Added the ability to send paid media to any chat.
  - Added the parameter `business_connection_id` to the method *aiogram.methods.send_paid_media.SendPaidMedia*, allowing bots to send paid media on behalf of a business account.
  - Added the field `paid_media` to the class *aiogram.types.transaction_partner_user.TransactionPartnerUser* for transactions involving paid media.
  - Added the method *aiogram.methods.create_chat_subscription_invite_link.CreateChatSubscriptionInviteLink*, allowing bots to create subscription invite links.
  - Added the method *aiogram.methods.edit_chat_subscription_invite_link.EditChatSubscriptionInviteLink*, allowing bots to edit the name of subscription invite links.
  - Added the field `until_date` to the class *aiogram.types.chat_member_member.ChatMemberMember* for members with an active subscription.
  - Added support for paid reactions and the class *aiogram.types.reaction_type_paid.ReactionTypePaid*.

  #1560

Misc

- Improved performance of StatesGroup #1507

2.6.4 3.11.0 (2024-08-09)

Features

- Added full support of Bot API 7.8
  - Added the field `has_main_web_app` to the class *aiogram.types.user.User*, which is returned in the response to *aiogram.methods.get_me.GetMe*.
  - Added the parameter `business_connection_id` to the methods *aiogram.methods.pin_chat_message.PinChatMessage* and *aiogram.methods.unpin_chat_message.UnpinChatMessage*, allowing bots to manage pinned messages on behalf of a business account.

  #1551

Bugfixes

- Fixed URL path in the "Open" button at the "demo/sendMessage" endpoint in the web_app example. #1546

Misc

- Added method *aiogram.types.message.Message.as_reply_parameters()*. Replaced usage of the argument `reply_to_message_id` with `reply_parameters` in all Message reply methods. #1538
- Added aiohttp v3.10 ` support. #1548

2.6.5 3.10.0 (2024-07-07)

Features

- Added full support of Bot API 7.7
  - Added the class *aiogram.types.refunded_payment.RefundedPayment*, containing information about a refunded payment.
  - Added the field `refunded_payment` to the class *aiogram.types.message.Message*, describing a service message about a refunded payment.

  #1536

2.6.6 3.9.0 (2024-07-06)

Features

- Added ChatMember resolution tool and updated 2.x migration guide. #1525
- Added full support of Bot API 7.6
  - **Added the classes *aiogram.types.paid_media.PaidMedia*,**
    aiogram.types.paid_media_info.PaidMediaInfo, *aiogram.types.paid_media_preview.PaidMediaPreview*, *aiogram.types.paid_media_photo.PaidMediaPhoto* and *aiogram.types.paid_media_video.PaidMediaVideo*, containing information about paid media.
  - **Added the method *aiogram.methods.send_paid_media.SendPaidMedia***
    and the classes *aiogram.types.input_paid_media.InputPaidMedia*, *aiogram.types.input_paid_media_photo.InputPaidMediaPhoto* and *aiogram.types.input_paid_media_video.InputPaidMediaVideo*, to support sending paid media.
  - **Documented that the methods *aiogram.methods.copy_message.CopyMessage***
    and *aiogram.methods.copy_messages.CopyMessages* cannot be used to copy paid media.
  - **Added the field `can_send_paid_media` to the class**
    aiogram.types.chat_full_info.ChatFullInfo.
  - **Added the field `paid_media` to the classes**
    aiogram.types.message.Message and *aiogram.types.external_reply_info.ExternalReplyInfo*.
  - **Added the class**
    aiogram.types.transaction_partner_telegram_ads.TransactionPartnerTelegramAds, containing information about Telegram Star transactions involving the Telegram Ads Platform.

- **Added the field `invoice_payload` to the class**
  aiogram.types.transaction_partner_user.TransactionPartnerUser, containing the bot-specified invoice payload.
- Changed the default opening mode for Direct Link Mini Apps.
- **Added support for launching Web Apps via t.me link in the class**
  aiogram.types.menu_button_web_app.MenuButtonWebApp.
- Added the field `section_separator_color` to the class `ThemeParams`.

#1533

Bugfixes

- Fixed event context resolving for the callback query that is coming from the business account #1520

2.6.7 3.8.0 (2024-06-19)

Features

- Added utility to safely deserialize any Telegram object or method to a JSON-compatible object (dict). (*>> Read more*) #1450
- Added full support of Bot API 7.5
  - **Added the classes *aiogram.types.star_transactions.StarTransactions*,**
    aiogram.types.star_transaction.StarTransaction, *aiogram.types.transaction_partner.TransactionPartner* and *aiogram.types.revenue_withdrawal_state.RevenueWithdrawalState*, containing information about Telegram Star transactions involving the bot.
  - **Added the method *aiogram.methods.get_star_transactions.GetStarTransactions***
    that can be used to get the list of all Telegram Star transactions for the bot.
  - **Added support for callback buttons in**
    aiogram.types.inline_keyboard_markup.InlineKeyboardMarkup for messages sent on behalf of a business account.
  - **Added support for callback queries originating from a message sent**
    on behalf of a business account.
  - **Added the parameter `business_connection_id` to the methods**
    aiogram.methods.edit_message_text.EditMessageText, *aiogram.methods.edit_message_media.EditMessageMedia*, *aiogram.methods.edit_message_caption.EditMessageCaption*, *aiogram.methods.edit_message_live_location.EditMessageLiveLocation*, *aiogram.methods.stop_message_live_location.StopMessageLiveLocation* and *aiogram.methods.edit_message_reply_markup.EditMessageReplyMarkup*, allowing the bot to edit business messages.
  - **Added the parameter `business_connection_id` to the method**
    aiogram.methods.stop_poll.StopPoll, allowing the bot to stop polls it sent on behalf of a business account.

  #1518

Bugfixes

- Increased DNS cache ttl setting to aiohttp session as a workaround for DNS resolution issues in aiohttp. #1500

Improved Documentation

- Fixed MongoStorage section in the documentation by adding extra dependency to ReadTheDocs configuration. #1501
- Added information about dependency changes to the `2.x --> 3.x` migration guide. #1504

Misc

- [Only for contributors] Fail redis and mongo tests if incorrect URI provided + some storages tests refactoring

  If incorrect URIs provided to "–redis" and/or "–mongo" options tests should fail with errors instead of skipping. Otherwise the next scenario is possible:

  1) developer breaks RedisStorage and/or MongoStorage code
  2) tests are run with incorrect redis and/or mongo URIsprovided by "–redis" and "–mongo" options (for example, wrong port specified)
  3) tests pass because skipping doesn't fail tests run
  4) developer or reviewer doesn't notice that redis and/or mongo tests were skipped
  5) broken code gets in codebase

  Also some refactorings done (related with storages and storages tests). #1510

2.6.8 3.7.0 (2024-05-31)

Features

- Added new storage `aiogram.fsm.storage.MongoStorage` for Finite State Machine based on Mongo DB (using `motor` library) #1434
- Added full support of Bot API 7.4 #1498

Bugfixes

- Fixed wrong `MarkdownV2` custom emoji parsing in `aiogram.utils.text_decorations` #1496

Deprecations and Removals

- Removed deprecated arguments from Bot class `parse_mode`, `disable_web_page_preview`, `protect_content` as previously announced in v3.4.0. #1494

Misc

- Improved code consistency and readability in code examples by refactoring imports, adjusting the base webhook URL, modifying bot instance initialization to utilize DefaultBotProperties, and updating router message handlers. #1482

2.6.9 3.6.0 (2024-05-06)

Features

- Added full support of Bot API 7.3 #1480

Improved Documentation

- Added telegram objects transformation block in 2.x -> 3.x migration guide #1412

2.6.10 3.5.0 (2024-04-23)

Features

- Added **message_thread_id** parameter to **ChatActionSender** class methods. #1437
- Added context manager interface to Bot instance, from now you can use:

  ```
  async with Bot(...) as bot:
      ...
  ```

  instead of

  ```
  async with Bot(...).context() as bot:
      ...
  ```

  #1468

Bugfixes

-
  - **WebAppUser Class Fields**: Added missing *is_premium*, *added_to_attachment_menu*, and *allows_write_to_pm* fields to *WebAppUser* class to align with the Telegram API.
  - **WebAppChat Class Implementation**: Introduced the *WebAppChat* class with all its fields (*id*, *type*, *title*, *username*, and *photo_url*) as specified in the Telegram API, which was previously missing from the library.
  - **WebAppInitData Class Fields**: Included previously omitted fields in the *WebAppInitData* class: *chat*, *chat_type*, *chat_instance*, to match the official documentation for a complete Telegram Web Apps support.

  #1424
- Fixed poll answer FSM context by handling `voter_chat` for `poll_answer` event #1436
- Added missing error handling to `_background_feed_update` (when in `handle_in_background=True` webhook mode) #1458

Improved Documentation

- Added WebAppChat class to WebApp docs, updated uk_UA localisation of WebApp docs. #1433

Misc

- Added full support of Bot API 7.2 #1444
- Loosened pydantic version upper restriction from <2.7 to <2.8 #1460

2.6.11 3.4.1 (2024-02-17)

Bugfixes

- Fixed JSON serialization of the `LinkPreviewOptions` class while it is passed as bot-wide default options. #1418

2.6.12 3.4.0 (2024-02-16)

Features

- Reworked bot-wide globals like `parse_mode`, `disable_web_page_preview`, and others to be more flexible.

  Warning

  Note that the old way of setting these global bot properties is now deprecated and will be removed in the next major release.

  #1392
- A new enum `KeyboardButtonPollTypeType` for `KeyboardButtonPollTypeType.type` field has bed added. #1398
- Added full support of Bot API 7.1
  - Added support for the administrator rights `can_post_stories`, `can_edit_stories`, `can_delete_stories` in supergroups.
  - Added the class `ChatBoostAdded` and the field `boost_added` to the class `Message` for service messages about a user boosting a chat.
  - Added the field `sender_boost_count` to the class `Message`.
  - Added the field `reply_to_story` to the class `Message`.
  - Added the fields `chat` and `id` to the class `Story`.
  - Added the field `unrestrict_boost_count` to the class `Chat`.
  - Added the field `custom_emoji_sticker_set_name` to the class `Chat`.

  #1417

Bugfixes

- Update KeyboardBuilder utility, fixed type-hints for button method, adjusted limits of the different markup types to real world values. #1399
- Added new `reply_parameters` param to `message.send_copy` because it hasn't been added there #1403

Improved Documentation

- Add notion "Working with plural forms" in documentation Utils -> Translation #1395

2.6.13 3.3.0 (2023-12-31)

Features

- Added full support of Bot API 7.0
  - Reactions
  - Replies 2.0
  - Link Preview Customization
  - Block Quotation
  - Multiple Message Actions
  - Requests for multiple users
  - Chat Boosts
  - Giveaway
  - Other changes

  #1387

2.6.14 3.2.0 (2023-11-24)

Features

- Introduced Scenes feature that helps you to simplify user interactions using Finite State Machine. Read more about *Scenes*. #1280
- Added the new FSM strategy `CHAT_TOPIC`, which sets the state for the entire topic in the chat, also works in private messages and regular groups without topics. #1343

Bugfixes

- Fixed `parse_mode` argument in the in `Message.send_copy` shortcut. Disable by default. #1332
- Added ability to get handler flags from filters. #1360
- Fixed a situation where a `CallbackData` could not be parsed without a default value. #1368

Improved Documentation

- Corrected grammatical errors, improved sentence structures, translation for migration 2.x-3.x #1302
- Minor typo correction, specifically in module naming + some grammar. #1340
- Added *CITATION.cff* file for automatic academic citation generation. Now you can copy citation from the GitHub page and paste it into your paper. #1351
- Minor typo correction in middleware docs. #1353

Misc

- Fixed ResourceWarning in the tests, reworked `RedisEventsIsolation` fixture to use Redis connection from `RedisStorage` #1320
- Updated dependencies, bumped minimum required version:
  - `magic-filter` - fixed *.resolve* operation
  - `pydantic` - fixed compatibility (broken in 2.4)
  - `aiodns` - added new dependency to the `fast` extras (`pip install aiogram[fast]`)
  - *others…*

  #1327
- Prevent update handling task pointers from being garbage collected, backport from 2.x #1331
- Updated `typing-extensions` package version range in dependencies to fix compatibility with `FastAPI` #1347
- Introduce Python 3.12 support #1354
- Speeded up CallableMixin processing by caching references to nested objects and simplifying kwargs assembly. #1357
- Added `pydantic` v2.5 support. #1361
- Updated `thumbnail` fields type to `InputFile` only #1372

2.6.15 3.1.1 (2023-09-25)

Bugfixes

- Fixed *pydantic* version <2.4, since 2.4 has breaking changes. #1322

2.6.16 3.1.0 (2023-09-22)

Features

- Added support for custom encoders/decoders for payload (and also for deep-linking). #1262
- Added `aiogram.utils.input_media.MediaGroupBuilder` for media group construction. #1293
- Added full support of Bot API 6.9 #1319

Bugfixes

- Added actual param hints for *InlineKeyboardBuilder* and *ReplyKeyboardBuilder*. #1303
- Fixed priority of events isolation, now user state will be loaded only after lock is acquired #1317

2.6.17 3.0.0 (2023-09-01)

Bugfixes

- Replaced `datetime.datetime` with *DateTime* type wrapper across types to make dumped JSONs object more compatible with data that is sent by Telegram. #1277
- Fixed magic `.as_(...)` operation for values that can be interpreted as *False* (e.g. *0*). #1281
- Italic markdown from utils now uses correct decorators #1282
- Fixed method `Message.send_copy` for stickers. #1284
- Fixed `Message.send_copy` method, which was not working properly with stories, so not you can copy stories too (forwards messages). #1286
- Fixed error overlapping when validation error is caused by remove_unset root validator in base types and methods. #1290

2.6.18 3.0.0rc2 (2023-08-18)

Bugfixes

- Fixed missing message content types (`ContentType.USER_SHARED`, `ContentType.CHAT_SHARED`) #1252
- Fixed nested hashtag, cashtag and email message entities not being parsed correctly when these entities are inside another entity. #1259
- Moved global filters check placement into router to add chance to pass context from global filters into handlers in the same way as it possible in other places #1266

Improved Documentation

- Added error handling example *examples/error_handling.py* #1099
- Added a few words about skipping pending updates #1251
- Added a section on Dependency Injection technology #1253
- This update includes the addition of a multi-file bot example to the repository. #1254
- Refactored examples code to use aiogram enumerations and enhanced chat messages with markdown beautification's for a more user-friendly display. #1256
- Supplemented "Finite State Machine" section in Migration FAQ #1264
- Removed extra param in docstring of TelegramEventObserver's filter method and fixed typo in I18n documentation. #1268

Misc

- Enhanced the warning message in dispatcher to include a JSON dump of the update when update type is not known. #1269
- Added support for Bot API 6.8 #1275

2.6.19 3.0.0rc1 (2023-08-06)

Features

- Added Currency enum. You can use it like this:

  ```
  from aiogram.enums import Currency

  await bot.send_invoice(
      ...,
      currency=Currency.USD,
      ...
  )
  ```

  #1194

- Updated keyboard builders with new methods for integrating buttons and keyboard creation more seamlessly. Added functionality to create buttons from existing markup and attach another builder. This improvement aims to make the keyboard building process more user-friendly and flexible. #1236
- Added support for message_thread_id in ChatActionSender #1249

Bugfixes

- Fixed polling startup when "bot" key is passed manually into dispatcher workflow data #1242
- Added codegen configuration for lost shortcuts:
  - ShippingQuery.answer
  - PreCheckoutQuery.answer
  - Message.delete_reply_markup

  #1244

Improved Documentation

- Added documentation for webhook and polling modes. #1241

Misc

- Reworked InputFile reading, removed `__aiter__` method, added *bot: Bot* argument to the `.read(...)` method, so, from now URLInputFile can be used without specifying bot instance. #1238
- Code-generated `__init__` typehints in types and methods to make IDE happy without additional pydantic plugin #1245

2.6.20 3.0.0b9 (2023-07-30)

Features

- Added new shortcuts for *aiogram.types.chat_member_updated.ChatMemberUpdated* to send message to chat that member joined/left. #1234
- Added new shortcuts for *aiogram.types.chat_join_request.ChatJoinRequest* to make easier access to sending messages to users who wants to join to chat. #1235

Bugfixes

- Fixed bot assignment in the `Message.send_copy` shortcut #1232
- Added model validation to remove UNSET before field validation. This change was necessary to correctly handle parse_mode where 'UNSET' is used as a sentinel value. Without the removal of 'UNSET', it would create issues when passed to model initialization from Bot.method_name. 'UNSET' was also added to typing. #1233
- Updated pydantic to 2.1 with few bugfixes

Improved Documentation

- Improved docs, added basic migration guide (will be expanded later) #1143

Deprecations and Removals

- Removed the use of the context instance (Bot.get_current) from all placements that were used previously. This is to avoid the use of the context instance in the wrong place. #1230

2.6.21 3.0.0b8 (2023-07-17)

Features

- Added possibility to use custom events in routers (If router does not support custom event it does not break and passes it to included routers). #1147
- Added support for FSM in Forum topics.

  The strategy can be changed in dispatcher:

  ```
  from aiogram.fsm.strategy import FSMStrategy
  ...
  dispatcher = Dispatcher(
      fsm_strategy=FSMStrategy.USER_IN_TOPIC,
      storage=...,  # Any persistent storage
  )
  ```

  Note

  If you have implemented you own storages you should extend record key generation with new one attribute - `thread_id`

  #1161
- Improved CallbackData serialization.
  - Minimized UUID (hex without dashes)
  - Replaced bool values with int (true=1, false=0)

  #1163
- Added a tool to make text formatting flexible and easy. More details on the *corresponding documentation page* #1172
- Added `X-Telegram-Bot-Api-Secret-Token` header check #1173
- Made `allowed_updates` list to revolve automatically in start_polling method if not set explicitly. #1178
- Added possibility to pass custom headers to `URLInputFile` object #1191

Bugfixes

- Change type of result in InlineQueryResult enum for `InlineQueryResultCachedMpeg4Gif` and `InlineQueryResultMpeg4Gif` to more correct according to documentation.

  Change regexp for entities parsing to more correct (`InlineQueryResultType.yml`). #1146
- Fixed signature of startup/shutdown events to include the `**dispatcher.workflow_data` as the handler arguments. #1155
- Added missing `FORUM_TOPIC_EDITED` value to content_type property #1160
- Fixed compatibility with Python 3.8-3.9 (from previous release) #1162
- Fixed the markdown spoiler parser. #1176
- Fixed workflow data propagation #1196
- Fixed the serialization error associated with nested subtypes like InputMedia, ChatMember, etc.

  The previously generated code resulted in an invalid schema under pydantic v2, which has stricter type parsing. Hence, subtypes without the specification of all subtype unions were generating an empty object. This has been rectified now. #1213

Improved Documentation

- Changed small grammar typos for `upload_file` #1133

Deprecations and Removals

- Removed text filter in due to is planned to remove this filter few versions ago.

  Use `F.text` instead #1170

Misc

- Added full support of Bot API 6.6

  > **Danger**
  >
  > Note that this issue has breaking changes described in the Bot API changelog, this changes is not breaking in the API but breaking inside aiogram because Beta stage is not finished.

  #1139
- Added full support of Bot API 6.7

  > **Warning**
  >
  > Note that arguments *switch_pm_parameter* and *switch_pm_text* was deprecated and should be changed to *button* argument as described in API docs.

  #1168

- Updated Pydantic to V2

  Warning

  Be careful, not all libraries is already updated to using V2

  #1202

- Added global defaults `disable_web_page_preview` and `protect_content` in addition to `parse_mode` to the Bot instance, reworked internal request builder mechanism. #1142
- Removed bot parameters from storages #1144
- Replaced ContextVar's with a new feature called Validation Context in Pydantic to improve the clarity, usability, and versatility of handling the Bot instance within method shortcuts.

  Danger

  Breaking: The 'bot' argument now is required in *URLInputFile*

  #1210

- Updated magic-filter with new features
  - Added hint for `len(F)` error
  - Added not in operation

  #1221

2.6.22 3.0.0b7 (2023-02-18)

Warning

Note that this version has incompatibility with Python 3.8-3.9 in case when you create an instance of Dispatcher outside of the any coroutine.

Sorry for the inconvenience, it will be fixed in the next version.

This code will not work:

```
dp = Dispatcher()

def main():
    ...
    dp.run_polling(...)

main()
```

But if you change it like this it should works as well:

```
router = Router()

async def main():
    dp = Dispatcher()
    dp.include_router(router)
    ...
    dp.start_polling(...)

asyncio.run(main())
```

Features

- Added missing shortcuts, new enums, reworked old stuff

  Breaking All previously added enums is re-generated in new place - *aiogram.enums* instead of *aiogram.types*

  Added enums: `aiogram.enums.bot_command_scope_type.BotCommandScopeType`, `aiogram.enums.chat_action.ChatAction`, `aiogram.enums.chat_member_status.ChatMemberStatus`, `aiogram.enums.chat_type.ChatType`, `aiogram.enums.content_type.ContentType`, `aiogram.enums.dice_emoji.DiceEmoji`, `aiogram.enums.inline_query_result_type.InlineQueryResultType`, `aiogram.enums.input_media_type.InputMediaType`, `aiogram.enums.mask_position_point.MaskPositionPoint`, `aiogram.enums.menu_button_type.MenuButtonType`, `aiogram.enums.message_entity_type.MessageEntityType`, `aiogram.enums.parse_mode.ParseMode`, `aiogram.enums.poll_type.PollType`, `aiogram.enums.sticker_type.StickerType`, `aiogram.enums.topic_icon_color.TopicIconColor`, `aiogram.enums.update_type.UpdateType`,

  Added shortcuts:

  – *Chat* `aiogram.types.chat.Chat.get_administrators()`, `aiogram.types.chat.Chat.delete_message()`, `aiogram.types.chat.Chat.revoke_invite_link()`, `aiogram.types.chat.Chat.edit_invite_link()`, `aiogram.types.chat.Chat.create_invite_link()`, `aiogram.types.chat.Chat.export_invite_link()`, `aiogram.types.chat.Chat.do()`, `aiogram.types.chat.Chat.delete_sticker_set()`, `aiogram.types.chat.Chat.set_sticker_set()`, `aiogram.types.chat.Chat.get_member()`, `aiogram.types.chat.Chat.get_member_count()`, `aiogram.types.chat.Chat.leave()`, `aiogram.types.chat.Chat.unpin_all_messages()`, `aiogram.types.chat.Chat.unpin_message()`, `aiogram.types.chat.Chat.pin_message()`, `aiogram.types.chat.Chat.set_administrator_custom_title()`, `aiogram.types.chat.Chat.set_permissions()`, `aiogram.types.chat.Chat.promote()`, `aiogram.types.chat.Chat.restrict()`, `aiogram.types.chat.Chat.unban()`, `aiogram.types.chat.Chat.ban()`, `aiogram.types.chat.Chat.set_description()`, `aiogram.types.chat.Chat.set_title()`, `aiogram.types.chat.Chat.delete_photo()`, `aiogram.types.chat.Chat.set_photo()`,

  – *Sticker*: `aiogram.types.sticker.Sticker.set_position_in_set()`, `aiogram.types.sticker.Sticker.delete_from_set()`,

  – *User*: `aiogram.types.user.User.get_profile_photos()`

  #952

- Added *callback answer* feature #1091
- Added a method that allows you to compactly register routers #1117

Bugfixes

- Check status code when downloading file #816
- Fixed *ignore_case* parameter in `aiogram.filters.command.Command` filter #1106

Misc

- Added integration with new code-generator named Butcher #1069
- Added full support of Bot API 6.4 #1088
- Updated package metadata, moved build internals from Poetry to Hatch, added contributing guides. #1095
- Added full support of Bot API 6.5

  Danger

  Note that `aiogram.types.chat_permissions.ChatPermissions` is updated without backward compatibility, so now this object has no `can_send_media_messages` attribute

  #1112
- Replaced error `TypeError: TelegramEventObserver.__call__() got an unexpected keyword argument '<name>'` with a more understandable one for developers and with a link to the documentation. #1114
- Added possibility to reply into webhook with files #1120
- Reworked graceful shutdown. Added method to stop polling. Now polling started from dispatcher can be stopped by signals gracefully without errors (on Linux and Mac). #1124

2.6.23 3.0.0b6 (2022-11-18)

Features

- (again) Added possibility to combine filters with an *and/or* operations.

  Read more in "*Combining filters*" documentation section #1018
- Added following methods to `Message` class:
  - `Message.forward(...)`
  - `Message.edit_media(...)`
  - `Message.edit_live_location(...)`
  - `Message.stop_live_location(...)`
  - `Message.pin(...)`
  - `Message.unpin()`

  #1030
- Added following methods to `User` class:
  - `User.mention_markdown(...)`
  - `User.mention_html(...)`

#1049

- Added full support of Bot API 6.3 #1057

Bugfixes

- Fixed `Message.send_invoice` and `Message.reply_invoice`, added missing arguments #1047
- Fixed copy and forward in:
  - `Message.answer(...)`
  - `Message.copy_to(...)`

  #1064

Improved Documentation

- Fixed UA translations in index.po #1017
- Fix typehints for `Message`, `reply_media_group` and `answer_media_group` methods #1029
- Removed an old now non-working feature #1060

Misc

- Enabled testing on Python 3.11 #1044
- Added a mandatory dependency `certifi` in due to in some cases on systems that doesn't have updated ca-certificates the requests to Bot API fails with reason `[SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed:  self signed certificate in certificate chain` #1066

2.6.24 3.0.0b5 (2022-10-02)

Features

- Add PyPy support and run tests under PyPy #985
- Added message text to aiogram exceptions representation #988
- Added warning about using magic filter from *magic_filter* instead of *aiogram*'s ones. Is recommended to use *from aiogram import F* instead of *from magic_filter import F* #990
- Added more detailed error when server response can't be deserialized. This feature will help to debug unexpected responses from the Server #1014

Bugfixes

- Reworked error event, introduced `aiogram.types.error_event.ErrorEvent` object. #898
- Fixed escaping markdown in *aiogram.utils.markdown* module #903
- Fixed polling crash when Telegram Bot API raises HTTP 429 status-code. #995
- Fixed empty mention in command parsing, now it will be None instead of an empty string #1013

Improved Documentation

- Initialized Docs translation (added Ukrainian language) #925

Deprecations and Removals

- Removed filters factory as described in corresponding issue. #942

Misc

- Now Router/Dispatcher accepts only keyword arguments. #982

2.6.25 3.0.0b4 (2022-08-14)

Features

- Add class helper ChatAction for constants that Telegram BotAPI uses in sendChatAction request. In my opinion, this will help users and will also improve compatibility with 2.x version where similar class was called "ChatActions". #803
- Added possibility to combine filters or invert result

  Example:

  ```
  Text(text="demo") | Command(commands=["demo"])
  MyFilter() & AnotherFilter()
  ~StateFilter(state='my-state')
  ```

  #894
- Fixed type hints for redis TTL params. #922
- Added *full_name* shortcut for *Chat* object #929

Bugfixes

- Fixed false-positive coercing of Union types in API methods #901
- Added 3 missing content types:
  - proximity_alert_triggered
  - supergroup_chat_created
  - channel_chat_created

#906

- Fixed the ability to compare the state, now comparison to copy of the state will return *True*. #927
- Fixed default lock kwargs in RedisEventIsolation. #972

Misc

- Restrict including routers with strings #896
- Changed CommandPatterType to CommandPatternType in *aiogram/dispatcher/filters/command.py* #907
- Added full support of Bot API 6.1 #936
- **Breaking!** More flat project structure

  These packages was moved, imports in your code should be fixed:

  - `aiogram.dispatcher.filters` -> `aiogram.filters`
  - `aiogram.dispatcher.fsm` -> `aiogram.fsm`
  - `aiogram.dispatcher.handler` -> `aiogram.handler`
  - `aiogram.dispatcher.webhook` -> `aiogram.webhook`
  - `aiogram.dispatcher.flags/*` -> `aiogram.dispatcher.flags` (single module instead of package)

  #938
- Removed deprecated `router.<event>_handler` and `router.register_<event>_handler` methods. #941
- Deprecated filters factory. It will be removed in next Beta (3.0b5) #942
- *MessageEntity* method *get_text* was removed and *extract* was renamed to *extract_from* #944
- Added full support of Bot API 6.2 #975

2.6.26 3.0.0b3 (2022-04-19)

Features

- Added possibility to get command magic result as handler argument #889
- Added full support of Telegram Bot API 6.0 #890

Bugfixes

- Fixed I18n lazy-proxy. Disabled caching. #839
- Added parsing of spoiler message entity #865
- Fixed default *parse_mode* for *Message.copy_to()* method. #876
- Fixed CallbackData factory parsing IntEnum's #885

Misc

- Added automated check that pull-request adds a changes description to **CHANGES** directory #873
- Changed `Message.html_text` and `Message.md_text` attributes behaviour when message has no text. The empty string will be used instead of raising error. #874
- Used *redis-py* instead of *aioredis* package in due to this packages was merged into single one #882
- Solved common naming problem with middlewares that confusing too much developers - now you can't see the *middleware* and *middlewares* attributes at the same point because this functionality encapsulated to special interface. #883

2.6.27 3.0.0b2 (2022-02-19)

Features

- Added possibility to pass additional arguments into the aiohttp webhook handler to use this arguments inside handlers as the same as it possible in polling mode. #785
- Added possibility to add handler flags via decorator (like *pytest.mark* decorator but *aiogram.flags*) #836
- Added `ChatActionSender` utility to automatically sends chat action while long process is running.

  It also can be used as message middleware and can be customized via `chat_action` flag. #837

Bugfixes

- Fixed unexpected behavior of sequences in the StateFilter. #791
- Fixed exceptions filters #827

Misc

- Logger name for processing events is changed to `aiogram.events`. #830
- Added full support of Telegram Bot API 5.6 and 5.7 #835
- **BREAKING** Events isolation mechanism is moved from FSM storages to standalone managers #838

2.6.28 3.0.0b1 (2021-12-12)

Features

- Added new custom operation for MagicFilter named `as_`

  Now you can use it to get magic filter result as handler argument

  ```python
  from aiogram import F

  ...

  @router.message(F.text.regexp(r"^(\d+)$").as_("digits"))
  async def any_digits_handler(message: Message, digits: Match[str]):
      await message.answer(html.quote(str(digits)))
  ```

(continues on next page)

(continued from previous page)

```
@router.message(F.photo[-1].as_("photo"))
async def download_photos_handler(message: Message, photo: PhotoSize, bot: Bot):
    content = await bot.download(photo)
```

#759

Bugfixes

- Fixed: Missing `ChatMemberHandler` import in `aiogram/dispatcher/handler` #751

Misc

- Check `destiny` in case of no `with_destiny` enabled in RedisStorage key builder #776
- Added full support of Bot API 5.5 #777
- Stop using feature from #336. From now settings of client-session should be placed as initializer arguments instead of changing instance attributes. #778
- Make TelegramAPIServer files wrapper in local mode bi-directional (server-client, client-server) Now you can convert local path to server path and server path to local path. #779

2.6.29 3.0.0a18 (2021-11-10)

Features

- Breaking: Changed the signature of the session middlewares Breaking: Renamed AiohttpSession.make_request method parameter from call to method to match the naming in the base class Added middleware for logging outgoing requests #716
- Improved description of filters resolving error. For example when you try to pass wrong type of argument to the filter but don't know why filter is not resolved now you can get error like this:

```
aiogram.exceptions.FiltersResolveError: Unknown keyword filters: {'content_types'}
  Possible cases:
  - 1 validation error for ContentTypesFilter
    content_types
      Invalid content types {'42'} is not allowed here (type=value_error)
```

  #717
- **Breaking internal API change** Reworked FSM Storage record keys propagation #723
- Implemented new filter named `MagicData(magic_data)` that helps to filter event by data from middlewares or other filters

  For example your bot is running with argument named `config` that contains the application config then you can filter event by value from this config:

```
@router.message(magic_data=F.event.from_user.id == F.config.admin_id)
...
```

#724

Bugfixes

- Fixed I18n context inside error handlers #726
- Fixed bot session closing before emit shutdown #734
- Fixed: bound filter resolving does not require children routers #736

Misc

- Enabled testing on Python 3.10 Removed *async_lru* dependency (is incompatible with Python 3.10) and replaced usage with protected property #719
- Converted README.md to README.rst and use it as base file for docs #725
- Rework filters resolving:
  - Automatically apply Bound Filters with default values to handlers
  - Fix data transfer from parent to included routers filters

  #727
- Added full support of Bot API 5.4 https://core.telegram.org/bots/api-changelog#november-5-2021 #744

2.6.30 3.0.0a17 (2021-09-24)

Misc

- Added `html_text` and `md_text` to Message object #708
- Refactored I18n, added context managers for I18n engine and current locale #709

2.6.31 3.0.0a16 (2021-09-22)

Features

- Added support of local Bot API server files downloading

  When Local API is enabled files can be downloaded via *bot.download*/*bot.download_file* methods. #698
- Implemented I18n & L10n support #701

Misc

- Covered by tests and docs KeyboardBuilder util #699
- **Breaking!!!**. Refactored and renamed exceptions.
  - Exceptions module was moved from `aiogram.utils.exceptions` to `aiogram.exceptions`
  - Added prefix *Telegram* for all error classes

  #700
- Replaced all `pragma:  no cover` marks via global `.coveragerc` config #702

- Updated dependencies.

  Breaking for framework developers Now all optional dependencies should be installed as extra: *poetry install -E fast -E redis -E proxy -E i18n -E docs* #703

2.6.32 3.0.0a15 (2021-09-10)

Features

- Ability to iterate over all states in StatesGroup. Aiogram already had in check for states group so this is relative feature. #666

Bugfixes

- Fixed incorrect type checking in the `aiogram.utils.keyboard.KeyboardBuilder` #674

Misc

- Disable ContentType filter by default #668
- Moved update type detection from Dispatcher to Update object #669
- Updated **pre-commit** config #681
- Reworked **handlers_in_use** util. Function moved to Router as method **.resolve_used_update_types()** #682

2.6.33 3.0.0a14 (2021-08-17)

Features

- add aliases for edit/delete reply markup to Message #662
- Reworked outer middleware chain. Prevent to call many times the outer middleware for each nested router #664

Bugfixes

- Prepare parse mode for InputMessageContent in AnswerInlineQuery method #660

Improved Documentation

- Added integration with `towncrier` #602

Misc

- Added *.editorconfig* #650
- Redis storage speedup globals #651
- add allow_sending_without_reply param to Message reply aliases #663

2.6.34 2.14.3 (2021-07-21)

- Fixed `ChatMember` type detection via adding customizable object serialization mechanism (#624, #623)

2.6.35 2.14.2 (2021-07-26)

- Fixed `MemoryStorage` cleaner (#619)
- Fixed unused default locale in `I18nMiddleware` (#562, #563)

2.6.36 2.14 (2021-07-27)

- Full support of Bot API 5.3 (#610, #614)
- Fixed `Message.send_copy` method for polls (#603)
- Updated pattern for `GroupDeactivated` exception (#549
- Added `caption_entities` field in `InputMedia` base class (#583)
- Fixed HTML text decorations for tag `pre` (#597 fixes issues #596 and #481)
- Fixed `Message.get_full_command` method for messages with caption (#576)
- Improved `MongoStorage`: remove documents with empty data from `aiogram_data` collection to save memory. (#609)

2.6.37 2.13 (2021-04-28)

- Added full support of Bot API 5.2 (#572)
- Fixed usage of `provider_data` argument in `sendInvoice` method call
- Fixed builtin command filter args (#556) (#558)
- Allowed to use State instances FSM storage directly (#542)
- Added possibility to get i18n locale without User instance (#546)
- Fixed returning type of `Bot.*_chat_invite_link()` methods #548 (#549)
- Fixed deep-linking util (#569)
- Small changes in documentation - describe limits in docstrings corresponding to the current limit. (#565)
- Fixed internal call to deprecated 'is_private' method (#553)
- Added possibility to use `allowed_updates` argument in Polling mode (#564)

2.6.38 2.12.1 (2021-03-22)

- Fixed `TypeError: Value should be instance of 'User' not 'NoneType'` (#527)
- Added missing `Chat.message_auto_delete_time` field (#535)
- Added `MediaGroup` filter (#528)
- Added `Chat.delete_message` shortcut (#526)
- Added mime types parsing for `aiogram.types.Document` (#431)
- Added warning in `TelegramObject.__setitem__` when Telegram adds a new field (#532)
- Fixed `examples/chat_type_filter.py` (#533)
- Removed redundant definitions in framework code (#531)

2.6.39 2.12 (2021-03-14)

- Full support for Telegram Bot API 5.1 (#519)
- Fixed sending playlist of audio files and documents (#465, #468)
- Fixed `FSMContextProxy.setdefault` method (#491)
- Fixed `Message.answer_location` and `Message.reply_location` unable to send live location (#497)
- Fixed `user_id` and `chat_id` getters from the context at Dispatcher `check_key`, `release_key` and `throttle` methods (#520)
- Fixed `Chat.update_chat` method and all similar situations (#516)
- Fixed `MediaGroup` attach methods (#514)
- Fixed state filter for inline keyboard query callback in groups (#508, #510)
- Added missing `ContentTypes.DICE` (#466)
- Added missing vcard argument to `InputContactMessageContent` constructor (#473)
- Add missing exceptions: `MessageIdInvalid`, `CantRestrictChatOwner` and `UserIsAnAdministratorOfTheChat` (#474, #512)
- Added `answer_chat_action` to the `Message` object (#501)
- Added dice to `message.send_copy` method (#511)
- Removed deprecation warning from `Message.send_copy`
- Added an example of integration between externally created aiohttp Application and aiogram (#433)
- Added `split_separator` argument to `safe_split_text` (#515)
- Fixed some typos in docs and examples (#489, #490, #498, #504, #514)

2.6.40 2.11.2 (2021-11-10)

- Fixed default parse mode
- Added missing "supports_streaming" argument to answer_video method #462

2.6.41 2.11.1 (2021-11-10)

- Fixed files URL template
- Fix MessageEntity serialization for API calls #457
- When entities are set, default parse_mode become disabled (#461)
- Added parameter supports_streaming to reply_video, remove redundant docstrings (#459)
- Added missing parameter to promoteChatMember alias (#458)

2.6.42 2.11 (2021-11-08)

- Added full support of Telegram Bot API 5.0 (#454)
- **Added possibility to more easy specify custom API Server (example)**
  - WARNING: API method `close` was named in Bot class as close_bot in due to Bot instance already has method with the same name. It will be changed in `aiogram 3.0`
- Added alias to Message object `Message.copy_to` with deprecation of `Message.send_copy`
- `ChatType.SUPER_GROUP` renamed to `ChatType.SUPERGROUP` (#438)

2.6.43 2.10.1 (2021-09-14)

- Fixed critical bug with getting asyncio event loop in executor. (#424) `AttributeError: 'NoneType' object has no attribute 'run_until_complete'`

2.6.44 2.10 (2021-09-13)

- Breaking change: Stop using _MainThread event loop in bot/dispatcher instances (#397)
- Breaking change: Replaced aiomongo with motor (#368, #380)
- Fixed: TelegramObject's aren't destroyed after update handling #307 (#371)
- Add setting current context of Telegram types (#369)
- Fixed markdown escaping issues (#363)
- Fixed HTML characters escaping (#409)
- Fixed italic and underline decorations when parse entities to Markdown
- Fixed #413: parse entities positioning (#414)
- Added missing thumb parameter (#362)
- Added public methods to register filters and middlewares (#370)
- Added ChatType builtin filter (#356)

- Fixed IDFilter checking message from channel (#376)
- Added missed answer_poll and reply_poll (#384)
- Added possibility to ignore message caption in commands filter (#383)
- Fixed addStickerToSet method
- Added preparing thumb in send_document method (#391)
- Added exception MessageToPinNotFound (#404)
- Fixed handlers parameter-spec solving (#408)
- Fixed CallbackQuery.answer() returns nothing (#420)
- CHOSEN_INLINE_RESULT is a correct API-term (#415)
- Fixed missing attributes for Animation class (#422)
- Added missed emoji argument to reply_dice (#395)
- Added is_chat_creator method to ChatMemberStatus (#394)
- Added missed ChatPermissions to __all__ (#393)
- Added is_forward method to Message (#390)
- Fixed usage of deprecated is_private function (#421)

and many others documentation and examples changes:

- Updated docstring of RedisStorage2 (#423)
- Updated I18n example (added docs and fixed typos) (#419)
- A little documentation revision (#381)
- Added comments about correct errors_handlers usage (#398)
- Fixed typo rexex -> regex (#386)
- Fixed docs Quick start page code blocks (#417)
- fixed type hints of callback_data (#400)
- Prettify readme, update downloads stats badge (#406)

2.6.45 2.9.2 (2021-06-13)

- Fixed `Message.get_full_command()` #352
- Fixed markdown util #353

2.6.46 2.9 (2021-06-08)

- Added full support of Telegram Bot API 4.9
- Fixed user context at poll_answer update (#322)
- Fix Chat.set_description (#325)
- Add lazy session generator (#326)
- Fix text decorations (#315, #316, #328)
- Fix missing `InlineQueryResultPhoto` `parse_mode` field (#331)

- Fix fields from parent object in `KeyboardButton` (#344 fixes #343)
- Add possibility to get bot id without calling `get_me` (#296)

2.6.47 2.8 (2021-04-26)

- Added full support of Bot API 4.8
- Added `Message.answer_dice` and `Message.reply_dice` methods (#306)

2.6.48 2.7 (2021-04-07)

- Added full support of Bot API 4.7 (#294 #289)
- Added default parse mode for send_animation method (#293 #292)
- Added new API exception when poll requested in public chats (#270)
- Make correct User and Chat get_mention methods (#277)
- Small changes and other minor improvements

2.6.49 2.6.1 (2021-01-25)

- Fixed reply `KeyboardButton` initializer with `request_poll` argument (#266)
- Added helper for poll types (`aiogram.types.PollType`)
- Changed behavior of Telegram_object `.as_*` and `.to_*` methods. It will no more mutate the object. (#247)

2.6.50 2.6 (2021-01-23)

- Full support of Telegram Bot API v4.6 (Polls 2.0) #265
- Aded new filter - IsContactSender (commit)
- Fixed proxy extra dependencies version #262

2.6.51 2.5.3 (2021-01-05)

- #255 Updated CallbackData factory validity check. More correct for non-latin symbols
- #256 Fixed `renamed_argument` decorator error
- #257 One more fix of CommandStart filter

2.6.52 2.5.2 (2021-01-01)

- Get back `quote_html` and `escape_md` functions

2.6.53 2.5.1 (2021-01-01)

- Hot-fix of `CommandStart` filter

2.6.54 2.5 (2021-01-01)

- Added full support of Telegram Bot API 4.5 (#250, #251)
- #239 Fixed `check_token` method
- #238, #241: Added deep-linking utils
- #248 Fixed support of aiohttp-socks
- Updated setup.py. No more use of internal pip API
- Updated links to documentations (https://docs.aiogram.dev)
- Other small changes and minor improvements (#223 and others…)

2.6.55 2.4 (2021-10-29)

- Added Message.send_copy method (forward message without forwarding)
- Safe close of aiohttp client session (no more exception when application is shutdown)
- No more "adWanced" words in project #209
- Arguments user and chat is renamed to user_id and chat_id in Dispatcher.throttle method #196
- Fixed set_chat_permissions #198
- Fixed Dispatcher polling task does not process cancellation #199, #201
- Fixed compatibility with latest asyncio version #200
- Disabled caching by default for lazy_gettext method of I18nMiddleware #203
- Fixed HTML user mention parser #205
- Added IsReplyFilter #210
- Fixed send_poll method arguments #211
- Added OrderedHelper #215
- Fix incorrect completion order. #217

2.6.56 2.3 (2021-08-16)

- Full support of Telegram Bot API 4.4
- Fixed #143
- Added new filters from issue #151: #172, #176, #182
- Added expire argument to RedisStorage2 and other storage fixes #145
- Fixed JSON and Pickle storages #138
- Implemented MongoStorage #153 based on aiomongo (soon motor will be also added)
- Improved tests
- Updated examples
- Warning: Updated auth widget util. #190
- Implemented throttle decorator #181

2.6.57 2.2 (2021-06-09)

- Provides latest Telegram Bot API (4.3)
- Updated docs for filters
- Added opportunity to use different bot tokens from single bot instance (via context manager, #100)
- IMPORTANT: Fixed Typo: `data` -> `bucket` in `update_bucket` for RedisStorage2 (#132)

2.6.58 2.1 (2021-04-18)

- Implemented all new features from Telegram Bot API 4.2
- `is_member` and `is_admin` methods of `ChatMember` and `ChatMemberStatus` was renamed to `is_chat_member` and `is_chat_admin`
- Remover func filter
- Added some useful Message edit functions (`Message.edit_caption`, `Message.edit_media`, `Message.edit_reply_markup`) (#121, #103, #104, #112)
- Added requests timeout for all methods (#110)
- Added `answer*` methods to `Message` object (#112)
- Maked some improvements of `CallbackData` factory
- Added deep-linking parameter filter to `CommandStart` filter
- Implemented opportunity to use DNS over socks (#97 -> #98)
- Implemented logging filter for extending LogRecord attributes (Will be usefull with external logs collector utils like GrayLog, Kibana and etc.)
- Updated `requirements.txt` and `dev_requirements.txt` files
- Other small changes and minor improvements

2.6.59 2.0.1 (2021-12-31)

- Implemented CallbackData factory (example)
- Implemented methods for answering to inline query from context and reply with animation to the messages. #85
- Fixed installation from tar.gz #84
- More exceptions (`ChatIdIsEmpty` and `NotEnoughRightsToRestrict`)

2.6.60 2.0 (2021-10-28)

This update will break backward compability with Python 3.6 and works only with Python 3.7+: - contextvars (PEP-567); - New syntax for annotations (PEP-563).

Changes: - Used contextvars instead of `aiogram.utils.context`; - Implemented filters factory; - Implemented new filters mechanism; - Allowed to customize command prefix in CommandsFilter; - Implemented mechanism of passing results from filters (as dicts) as kwargs in handlers (like fixtures in pytest); - Implemented states group feature; - Implemented FSM storage's proxy; - Changed files uploading mechanism; - Implemented pipe for uploading files from URL; - Implemented I18n Middleware; - Errors handlers now should accept only two arguments (current update and exception); - Used `aiohttp_socks` instead of `aiosocksy` for Socks4/5 proxy; - types.ContentType was divided to `types.ContentType` and `types.ContentTypes`; - Allowed to use rapidjson instead of ujson/json; - `.current()` method in bot and dispatcher objects was renamed to `get_current()`;

Full changelog - You can read more details about this release in migration FAQ: https://aiogram.readthedocs.io/en/latest/migration_1_to_2.html

2.6.61 1.4 (2021-08-03)

- Bot API 4.0 (#57)

2.6.62 1.3.3 (2021-07-16)

- Fixed markup-entities parsing;
- Added more API exceptions;
- Now InlineQueryResultLocation has live_period;
- Added more message content types;
- Other small changes and minor improvements.

2.6.63 1.3.2 (2021-05-27)

- Fixed crashing of polling process. (i think)
- Added parse_mode field into input query results according to Bot API Docs.
- Added new methods for Chat object. (#42, #43)
- **Warning**: disabled connections limit for bot aiohttp session.
- **Warning**: Destroyed "temp sessions" mechanism.
- Added new error types.
- Refactored detection of error type.

- Small fixes of executor util.
- Fixed RethinkDBStorage

2.6.64 1.3.1 (2018-05-27)

2.6.65 1.3 (2021-04-22)

- Allow to use Socks5 proxy (need manually install `aiosocksy`).
- Refactored `aiogram.utils.executor` module.
- **[Warning]** Updated requirements list.

2.6.66 1.2.3 (2018-04-14)

- Fixed API errors detection
- Fixed compability of `setup.py` with pip 10.0.0

2.6.67 1.2.2 (2018-04-08)

- Added more error types.
- Implemented method `InputFile.from_url(url: str)` for downloading files.
- Implemented big part of API method tests.
- Other small changes and mminor improvements.

2.6.68 1.2.1 (2018-03-25)

- Fixed handling Venue's [#27, #26]
- Added parse_mode to all medias (Bot API 3.6 support) [#23]
- Now regexp filter can be used with callback query data [#19]
- Improvements in `InlineKeyboardMarkup` & `ReplyKeyboardMarkup` objects [#21]
- Other bug & typo fixes and minor improvements.

2.6.69 1.2 (2018-02-23)

- Full provide Telegram Bot API 3.6
- Fixed critical error: `Fatal Python error: PyImport_GetModuleDict: no module dictionary!`
- Implemented connection pool in RethinkDB driver
- Typo fixes of documentstion
- Other bug fixes and minor improvements.

2.6.70 1.1 (2018-01-27)

- Added more methods for data types (like `message.reply_sticker(...)` or `file.download(...)`
- Typo fixes of documentstion
- Allow to set default parse mode for messages (`Bot( ... , parse_mode='HTML')`)
- Allowed to cancel event from the `Middleware.on_pre_process_<event type>`
- Fixed sending files with correct names.
- Fixed MediaGroup
- Added RethinkDB storage for FSM (`aiogram.contrib.fsm_storage.rethinkdb`)

2.6.71 1.0.4 (2018-01-10)

2.6.72 1.0.3 (2018-01-07)

- Added middlewares mechanism.
- Added example for middlewares and throttling manager.
- Added logging middleware (`aiogram.contrib.middlewares.logging.LoggingMiddleware`)
- Fixed handling errors in async tasks (marked as 'async_task')
- Small fixes and other minor improvements.

2.6.73 1.0.2 (2017-11-29)

2.6.74 1.0.1 (2017-11-21)

- Implemented `types.InputFile` for more easy sending local files
- **Danger!** Fixed typo in word pooling. Now whatever all methods with that word marked as deprecated and original methods is renamed to polling. Check it in you'r code before updating!
- Fixed helper for chat actions (`types.ChatActions`)
- Added example for media group.

2.6.75 1.0 (2017-11-19)

- Remaked data types serialozation/deserialization mechanism (Speed up).
- Fully rewrited all Telegram data types.
- Bot object was fully rewritted (regenerated).
- Full provide Telegram Bot API 3.4+ (with sendMediaGroup)
- Warning: Now `BaseStorage.close()` is awaitable! (FSM)
- Fixed compability with uvloop.
- More employments for `aiogram.utils.context`.
- Allowed to disable `ujson`.

- Other bug fixes and minor improvements.
- Migrated from Bitbucket to Github.

2.6.76 0.4.1 (2017-08-03)

2.6.77 0.4 (2017-08-05)

2.6.78 0.3.4 (2017-08-04)

2.6.79 0.3.3 (2017-07-05)

2.6.80 0.3.2 (2017-07-04)

2.6.81 0.3.1 (2017-07-04)

2.6.82 0.2b1 (2017-06-00)

2.6.83 0.1 (2017-06-03)

2.7 Contributing

You're welcome to contribute to aiogram!

aiogram is an open-source project, and anyone can contribute to it in any possible way

2.7.1 Developing

Before making any changes in the framework code, it is necessary to fork the project and clone the project to your PC and know how to do a pull-request.

How to work with pull-request you can read in the GitHub docs

Also in due to this project is written in Python, you will need Python to be installed (is recommended to use latest Python versions, but any version starting from 3.8 can be used)

Use virtualenv

You can create a virtual environment in a directory using `venv` module (it should be pre-installed by default):

This action will create a `.venv` directory with the Python binaries and then you will be able to install packages into that isolated environment.

Activate the environment

Linux / macOS:

```
source .venv/bin/activate
```

Windows cmd

```
.\.venv\Scripts\activate
```

Windows PowerShell

```
.\.venv\Scripts\activate.ps1
```

To check it worked, use described command, it should show the `pip` version and location inside the isolated environment

```
pip -V
```

Also make sure you have the latest pip version in your virtual environment to avoid errors on next steps:

```
python -m pip install --upgrade pip
```

Setup project

After activating the environment install *aiogram* from sources and their dependencies.

Linux / macOS:

```
pip install -e ."[dev,test,docs,fast,redis,mongo,proxy,i18n]"
```

Windows:

```
pip install -e .[dev,test,docs,fast,redis,mongo,proxy,i18n]
```

It will install `aiogram` in editable mode into your virtual environment and all dependencies.

Making changes in code

At this point you can make any changes in the code that you want, it can be any fixes, implementing new features or experimenting.

Format the code (code-style)

Note that this project is Black-formatted, so you should follow that code-style, too be sure You're correctly doing this let's reformat the code automatically:

```
black aiogram tests examples
isort aiogram tests examples
```

Run tests

All changes should be tested:

```
pytest tests
```

Also if you are doing something with Redis-storage or/and MongoDB-storage, you will need to test everything works with Redis or/and MongoDB:

```
pytest --redis redis://<host>:<port>/<db> --mongo mongodb://<user>:<password>@<host>:
↪<port> tests
```

Docs

We are using *Sphinx* to render docs in different languages, all sources located in *docs* directory, you can change the sources and to test it you can start live-preview server and look what you are doing:

```
sphinx-autobuild --watch aiogram/ docs/ docs/_build/
```

Docs translations

Translation of the documentation is very necessary and cannot be done without the help of the community from all over the world, so you are welcome to translate the documentation into different languages.

Before start, let's up to date all texts:

```
cd docs
make gettext
sphinx-intl update -p _build/gettext -l <language_code>
```

Change the `<language_code>` in example below to the target language code, after that you can modify texts inside `docs/locale/<language_code>/LC_MESSAGES` as `*.po` files by using any text-editor or specialized utilites for GNU Gettext, for example via poedit.

To view results:

```
sphinx-autobuild --watch aiogram/ docs/ docs/_build/ -D language=<language_code>
```

Describe changes

Describe your changes in one or more sentences so that bot developers know what's changed in their favorite framework - create *<code>.<category>.rst* file and write the description.

`<code>` is Issue or Pull-request number, after release link to this issue will be published to the *Changelog* page.

`<category>` is a changes category marker, it can be one of:

- `feature` - when you are implementing new feature
- `bugfix` - when you fix a bug
- `doc` - when you improve the docs
- `removal` - when you remove something from the framework
- `misc` - when changed something inside the Core or project configuration

If you have troubles with changing category feel free to ask Core-contributors to help with choosing it.

Complete

After you have made all your changes, publish them to the repository and create a pull request as mentioned at the beginning of the article and wait for a review of these changes.

2.7.2 Star on GitHub

You can "star" repository on GitHub - https://github.com/aiogram/aiogram (click the star button at the top right)

Adding stars makes it easier for other people to find this project and understand how useful it is.

2.7.3 Guides

You can write guides how to develop Bots on top of aiogram and publish it into YouTube, Medium, GitHub Books, any Courses platform or any other platform that you know.

This will help more people learn about the framework and learn how to use it

2.7.4 Take answers

The developers is always asks for any question in our chats or any other platforms like GitHub Discussions, StackOverflow and others, feel free to answer to this questions.

2.7.5 Funding

The development of the project is free and not financed by commercial organizations, it is my personal initiative (@JRootJunior) and I am engaged in the development of the project in my free time.

So, if you want to financially support the project, or, for example, give me a pizza or a beer, you can do it on OpenCollective.

PYTHON MODULE INDEX

INDEX