detache Documentation

Release 0.2.1

J Patrick Dill

Jun 13, 2018

Contents

Détaché is a framework for creating productive and efficient Discord bots, built off of discord.py.

With Détaché, you can easily create bots without sacrificing direct access to the API. Commands and similar features are split into groups called Plugins, allowing for better organization. It's inspired by the simplicity of Click and Flask.

Détaché's features include:

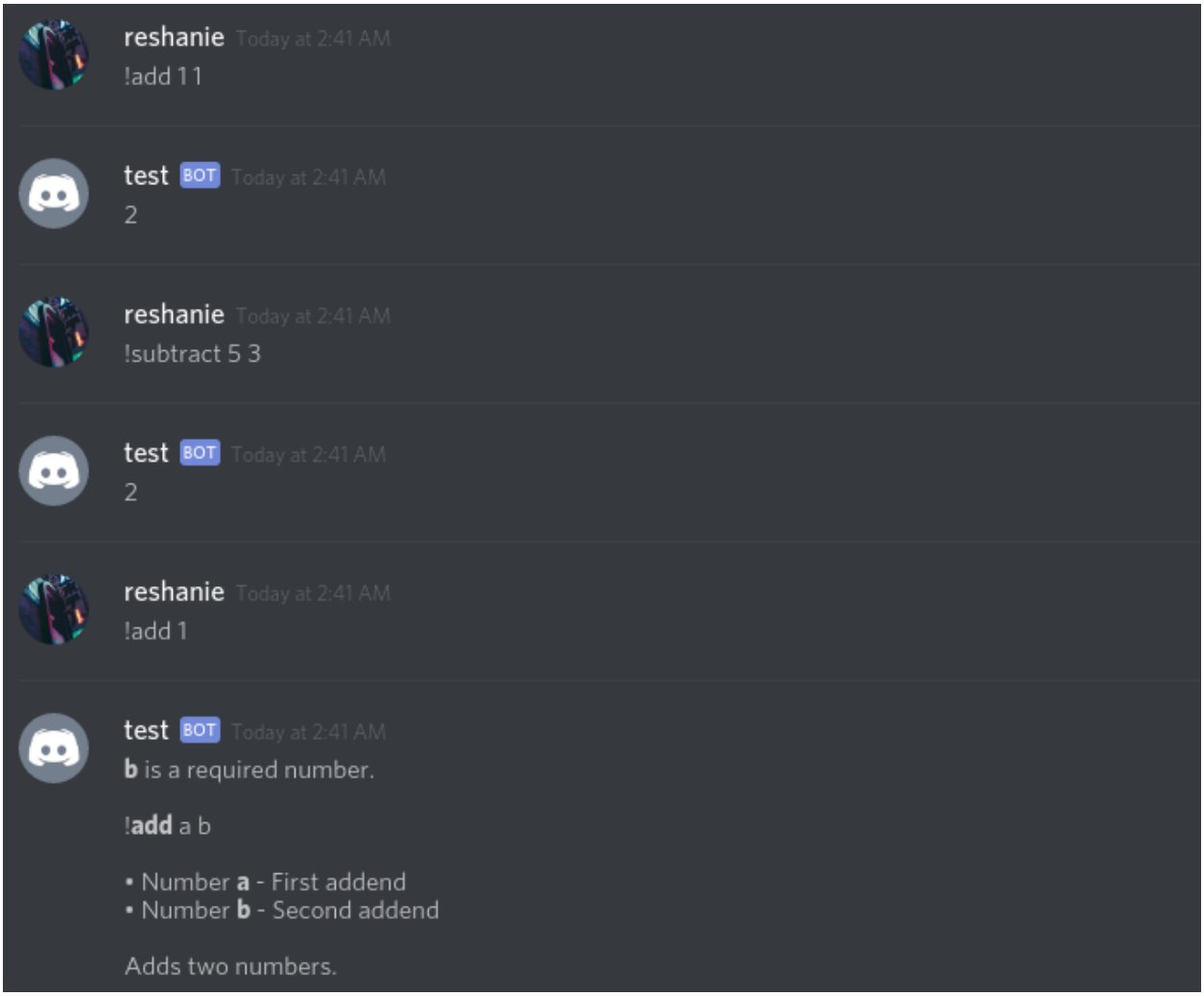
- intuitive argument parsing with support for custom types
- automatic documentation and help messages
- support for per-guild bot prefix via a callback
- support for sharding

Here's a simple bot that does math:

```
import detache

bot = detache.Bot(default_prefix="!")

@bot.plugin("Math")
class MathPlugin(detache.Plugin):
    """
    Basic math commands.
    """

    @detache.command("add", "Adds two numbers.")
    @detache.argument("a", detache.Number, help="First addend")
    @detache.argument("b", detache.Number, help="Second addend")
    async def add(self, ctx, a, b):
        return a + b

    @detache.command("subtract")
    @detache.argument("a", detache.Number, help="Minuend")
    @detache.argument("b", detache.Number, help="Subtrahend")
    async def subtract(self, ctx, a, b):
        """Subtracts two numbers."""

        return a - b
```

Both commands take two arguments, "a" and "b", which are specified as numbers. The commands return the sum or difference, which Détaché automatically replies with.

Commands can be documented with the command decorator, or by using docstrings.



If argument parsing fails, the generated documentation will be shown.

TODO: Plugin, command, and argument descriptions are shown in the automatically generated help message.

You can install the library from PyPI:

```
$ pip install detache
```

CHAPTER 1

Documentation

1.1 Quickstart

You can install Détaché from PyPI:

```
$ pip install detache
```

1.1.1 Basics

Détaché uses decorators to convert functions to commands.

Commands are declared through `detache.command()`, and arguments are added with *`detache.argument()`*.

```
@detache.command("hi", description="Says hi to someone")
@detache.argument("user", type=detache.User, help="User to say hi to")
async def say_hi(self, ctx, user):
    await ctx.send("Hi, " + user.mention + "!")
```

Command decorators are placed in front of the function and any arguments. The name passed will be a how a user calls the command, and an optional description can be passed as well.

Argument decorators are placed in the order they'll be used.

1.1.2 Plugins

Plugins are used to organize commands into groups. Plugins are created by inheriting a class from `detache.Plugin`. The `detache.Bot.plugin()` decorator should be placed before it with the name of the plugin.

```
import detache

bot = detache.Bot()
```

```
@bot.plugin("Example")
class ExamplePlugin(detache.Plugin)
    """Example Description used when documenting plugins and commands"""
```

Plugins can also be split across multiple files. For the plugin to work, you have to import and register it to the bot.

```
import detache

from plugins.example import ExamplePlugin

bot = detache.Bot()

bot.register_plugin(ExamplePlugin, name="Example")
```

1.1.3 Background Tasks

Détaché also supports background tasks, which are run when the bot connects to Discord. If the bot loses connection, the task will be cancelled and restart when the bot reconnects.

Similar to commands, the `detache.background_task()` decorator is used to declare a background task.

```
@detache.background_task("say_hi")
async def hi(self):
    # Say hi every 5 seconds.

    while True:
        for guild in self.bot.guilds:
            for channel in guild.channels:
                await channel.send("Hi!")

        await asyncio.sleep(5)
```

1.1.4 Event Listeners

An important feature of Détaché is that a provides a high level API while still giving access to the underlying events. Using event listeners, you can add a callback function to any discord.py event through the `detache.event_listener()` decorator.

This event listener listens for any "on_message_delete" events.

```
@detache.event_listener("on_message_delete")
async def undelete(self, message):
    await message.channel.send(
        "{} said {!r} at {}".format(message.author.mention, message.content, message.
↪created_at)
    )
```

1.2 Arguments

`detache.`**`argument`**(*name*, *type=None*, *default=None*, *required=True*, *nargs=1*, *help=None*)

Command argument.

Parameters

- **name** – Name of the argument. Should match one of the function's arguments.
- **type** – (Optional) Argument type. Leave as None to accept any type.
- **default** – (Optional) Default value.
- **required** – (Optional) Whether the argument is required. Defaults to True
- **nargs** – (Optional) Number of times the argument can occur. Defaults to 1. -1 allows unlimited arguments.
- **help** – (Optional) Argument description.

If nargs is anything other than 1, the parsed argument will be returned as a list.

Détaché supports several argument types:

- `detache.String` - String of text. Can be multi-word if surrounded with double quotes
- `detache.Number` - Integer or float
- `detache.User` - discord.py member object. Can be passed as a mention or username#1234
- `detache.Channel` - discord.py channel object.
- `detache.Role` - discord.py role object.

Custom types can also be created by inheriting from `detache.Any`. This type takes a hexadecimal number and converts it to an int, for example:

```
class Hex(detache.Any):
    # arguments are parsed using regex patterns
    pattern = "(0x)?[0-9a-f]+"

    @classmethod  # <- must be a classmethod
    def convert(cls, ctx, raw):
        # a context object and the raw argument are passed for conversion

        # remove 0x
        if raw.startswith("0x"):
            raw = raw[2:]

        return int(raw, base=16)  # return the converted argument
```

1.2.1 Variadic Arguments

Variadic arguments allow an argument to be passed a specific or unlimited number of times. This is set with the *nargs* parameter. If this is set to -1, an unlimited number of arguments is accepted.

Variadic arguments are passed to the underlying function as a list.

Example:

```
@detache.command("add", "Variadic argument test that adds numbers.")
@detache.argument("addends", detache.Number, nargs=-1, help="Addends")
async def add_cmd(self, ctx, addends):
    return sum(addends)
```

If *nargs* is -1, then setting *required=False* will allow the argument to not be passed at all. Otherwise, the argument must be passed at least once.

Index

A