
Enigma Documentation

Release 0.1.0.dev0

Enigma Contributors

May 12, 2018

Contents

1	Contents	3
1.1	License	3
1.2	Contributors	3
1.3	Contributing	4
1.4	Installation	4
1.5	User Manual	6
1.6	Developer Handbook	8
1.7	enigma	15
2	Indices and tables	17
	Python Module Index	19

This is the documentation of **Enigma**.

1.1 License

MIT License

Copyright (c) 2018 Unethical Discord

Permission **is** hereby granted, free of charge, to **any** person obtaining a copy of this software **and** associated documentation files (the "**Software**"), to deal **in** the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, **and/or** sell copies of the Software, **and** to permit persons to whom the Software **is** furnished to do so, subject to the following conditions:

The above copyright notice **and** this permission notice shall be included **in all** copies **or** substantial portions of the Software.

THE SOFTWARE IS PROVIDED "**AS IS**", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.2 Contributors

- Vivek Joshy <daegontaven@gmail.com>

1.3 Contributing

1.3.1 Style Guide

All pull requests to the python source must follow [PEP 8](#) conventions.

All methods and functions must be in snake_case and not camelCase. If a module is written in python, it must also conform to the 79 character limit.

1.3.2 Pull Requests

We follow [Github Flow](#) as our workflow when creating pull requests. It is a neater and easier way to manage changes. You are also responsible for writing tests(where applicable) if you are contributing to a core module. If we see an area of code that requires tests, then we will not accept the PR until you write a test for that area of code. Tests ensure long term stability.

Also note that there are CI checks in place. If any automated tests fail, please rework and resubmit your PR.

1.4 Installation

Warning: Enigma is pre-alpha and highly unstable. Do not use this in production as API changes will take place rapidly without notice.

1.4.1 Installing the bot

For Users

Todo: Add instructions for Docker, Kubernetes and pip wheel releases.

Follow the developer instructions for now and verify you have a working installation with this command.

```
$ enigma -V # This should show the current version.
enigma 0.1.0.dev0
```

For Developers

Installing from source

Enigma is very easy to install from source. First clone the latest development version from the master branch.

```
git clone https://github.com/UnethicalDiscord/Enigma.git
```

Since Enigma has a lot of dependencies, it is wise to install a virtualenv first. Please do not use [pipenv](#) however. It's incompatible with Enigma's dependencies and may cause more problems in the future. If you wish to submit a pull request to fix this problem please read more [here](#)

First let's make a virtualenv. So we have to install it first.

```
pip install virtualenv
```

Then create a new virtualenv within the repository. If you name it `venv` it won't get checked in.

```
cd Enigma/  
virtualenv venv
```

Now let's activate the virtual environment.

```
source venv/bin/activate
```

You should now see your terminal change to show you are you now using a virtual environment. Let's install the package dependencies now. This may take a while depending on your machine.

```
pip install -r requirements.txt
```

Now let's install it locally as an editable installation to make sure our changes get picked up.

```
pip install -e .
```

Additionally, if you need to write tests run this command.

```
pip install -e .[TESTS]
```

1.4.2 Configuring a database

Enigma does not come with it's own database. So we need to install and configure one to make sure Enigma works properly. Currently Enigma uses MongoDB to store all it's data.

You need to either setup one using Google Cloud like we are in production or set one up yourself on a bare metal server or a VPS. Either ways, it is outside the scope of this documentation for the most part.

You can read more about setting up a database by following the official MongoDB [documentation](#) Once you've successfully done this, we need to setup an administrator with superuser capabilities. You can read more about setting this up [here](#)

To do this, first we need to open up the `mongo` shell and create our user adjusting the commands below as needed.

```
use admin  
db.createUser(  
  {  
    user: "myUserAdmin",  
    pwd: "abc123",  
    roles: [ { role: "root", db: "admin" } ]  
  }  
)
```

This will create a superuser role giving Enigma complete control over the database. Take these credentials down as will you need to use them in the config file.

For Developers

Read the *Developer Handbook* to start contributing.

1.5 User Manual

1.5.1 Basic Usage Instructions

Configuring Enigma

To start using Enigma, we to get some configuration details. First let's make sure Enigma is installed.

```
$ enigma -V
enigma 0.1.0.dev
```

Looks good! So, we need to head over to the [discord developers portal](#) and create our bot user.

New App By creating an API application, you agree to the [Discord API Terms of Service](#)

APP NAME • Ping Bot
Something users will recognize and trust

REDIRECT URI(S) Add Redirect
You must specify at least one URI for authentication to work. If you pass a URI in an OAuth request, it must exactly match one of the URIs you enter here. [Learn more](#)

APP DESCRIPTION I will ping you back if you ask me to!
Maximum 400 characters

APP ICON Remove

Cancel Create App

Now this is the most important part. We need to create a configuration file which is also valid TOML.

```
[bot]
token = ""

[database]
host = []
```

(continues on next page)

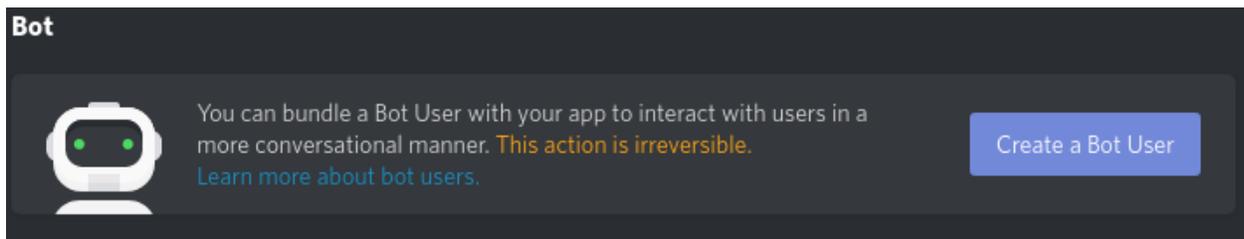
(continued from previous page)

```
port =
username = ""
password = ""
database = ""

[global]

prefixes = []
description = ""
```

To fill this out we need to know some details about our discord bot user. Simply scrolling down and clicking “Create a Bot User” will do the job.



Next click to reveal the token.

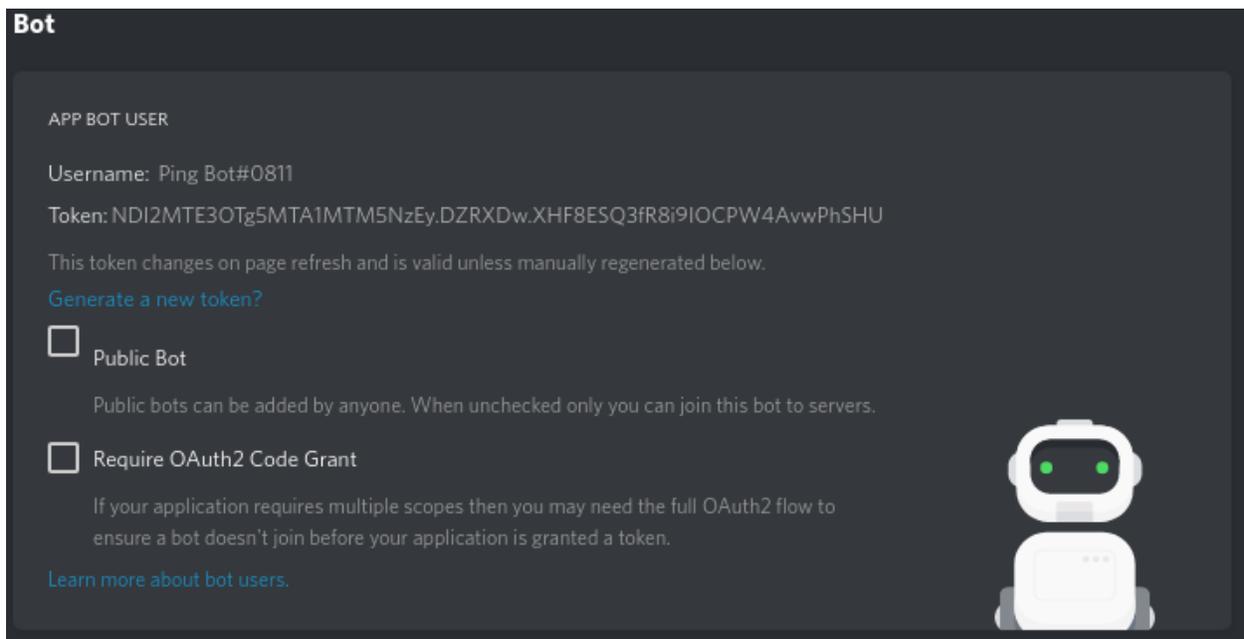
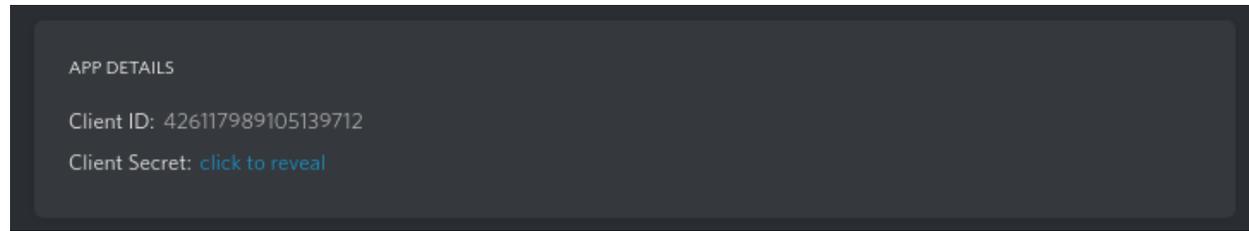


Fig. 1: Make sure to copy this token down!

For now we won't be delving in making our bot public and we'll stick to using our bot privately in a server of our choice. Save your changes and use this link replacing `BOT_CLIENT_ID` with your bot's client ID to invite Enigma to our server,

Invite Link : `https://discordapp.com/api/oauth2/authorize?client_id=BOT_CLIENT_ID&permissions=8&scope=bot`

You can get the client ID from your bot's app page.



Now that we have all the details, we can start filling in our config file. It should look something like this. You should also already have your database connection details. If not, read *Configuring a database*.

```
[bot]

token = "NDI2MTE3OTg5MTA1MTM5NzEy.DZRbXQ.CYHYtqRXjWYgJO9PqLoIv-HT8SE"

[database]

host = ["myvps.com"] # This can also be a list of hosts (including replica sets)
port = 27017
username = "pingbot"
password = "ilov3bacon"
database = "pingbot"
replica_set = "rs0" # This is needed added when using replica sets

[global]

prefixes = ["+", ">"]
description = "I will ping you back. Don't worry!"
```

Starting Enigma

Now that we have a config file ready. Let's save it is somewhere. By convention, it's named `app.cfg`.

Now let's tell Enigma to start by passing the path to this file as an argument.

```
enigma start --config /path/to/app.cfg
```

Note: You can also place it in the `enigma/app.cfg` folder of the repository if you installed from source. However, you must make sure to name it `app.cfg` in this case or Enigma will throw an error.

1.6 Developer Handbook

1.6.1 Introduction

Building a discord bot can sometimes be overwhelming if you need complex commands. Keeping track of logging, configuration files, databases, efficiency, sharding, devops and more. With Enigma you don't have to worry about the boring parts and focus on building your commands the way you need them.

Prerequisite Knowledge

Simple Commands

- You must have built a very basic bot using `discord.py`
- Basic experience with `asyncio`

Advanced Commands

- Experience using relational databases
- Knowledge of MongoDB (not mandatory unless you want to access the database directly)

Terminology

Before we begin building commands, there's some jargon you need to get familiar with.

Cogs:

Quoting `add_cog` in the `discord.py` documentation.

“A cog is a class that has its own event listeners and commands.

They are meant as a way to organize multiple relevant commands into a singular class that shares some state or no state at all.”

Extensions:

From the `discord.py` docs

“An extension is a python module that contains commands, cogs, or listeners.”

Plugins:

Plugins are simply extensions that have cogs with extra metadata and custom methods called in it. An existing cog can be converted into a plugin by defining a `plugin_data` variable in it's class. However, plugins are not guaranteed to work as a cog in another discord bot.

To give you a better picture:

- All plugins are extensions, but all extensions are not plugins.
- All cogs work with plugins, but not cogs built specifically for plugins.

Plugin Metadata:

This is simply a local `plugin_data` variable of type `dict` defined in a plugin. This defines metadata like the name, description, status and other properties of a plugin.

Plugin Setup:

This is a local `setup()` function defined outside the cog's class. It's used to do initialize cogs and prepare the plugin to be imported as an extension. Although it is fairly easy to initialize other cogs and commands directly from the `setup()` function, it's recommended to only place commands that complement each other into the same plugin.

Entities:

Entities are simply `discord data model` objects that represents anything that for which information can be stored. In simple terms, entities are any object in discord that can be referenced by an ID.

Entity States:

These are MongoDB database collections that can store data about a particular entity.

1.6.2 Building Plugins

Building a plugin in enigma is very easy as you will see soon. However, before we start, we need a working bot to build plugins for. If you haven't already, setup a bot by following *Basic Usage Instructions*.

All builtin plugins are stored in the `plugins` directory found in the root folder and grouped by categories in their respective folders.

Tutorials

Basic Tutorials

Intermediate Tutorials

Prerequisites

- You will need a MongoDB client like Robo3T or Compass to view collections
- An IDE or code editor like Pycharm, Atom, Sublime etc
- Minimal knowledge of collections, documents, databases and CRUD operations in MongoDB

Contents

Tags

Tags are commands used to save frequently sent messages like rules, instructions and help information in a guild. It's a very useful feature and implemented in a lot of bots on discord. So let's build one for our self.

Before we begin building our plugin, we need an idea of what our command will look like. We want users to be able to use newlines and any characters without sacrificing usability.

It will look something like this for a user in a guild:

```
+tag add rules

__**Chat Rules**__
**1.** An important rule
**2.** Another rule

__**Voice Chat Rules**__
**1.** Don't scream in voice chat
**2.** No trolling
```

Now that we know what we want our command to look like, let's create our plugin. Create a `tags.py` file in the `Plugins/` directory and put this code in it. You can remove the comments if you want.

Warning: There is an existing tags plugin in `Plugins/Utilities/tags.py`. You should move it to a safe location in another folder before continuing. You can't use two cogs with the same name at the same time.

```

import discord
from discord.ext import commands

plugin_data = {
    "name": "Tags"
}

class Tag:
    def __init__(self, bot):
        self.bot = bot
        self.data = plugin_data

        # Easier access to common variables
        self.logger = self.bot.logger

    @commands.command(
        name="tag"
    )
    @commands.guild_only() # Only allow usage from inside a guild
    async def tag(self, ctx, tag: str):
        """
        This will take a single parameter tag when someone uses the
        command +tag my_tag. Here my_tag will get passed to tag and
        ultimately to our logger.
        """
        self.logger.debug(f"Tag: {tag}")

```

Now *start enigma* and send a test command with a tag name to ensure everything works properly. If everything went smoothly, we need to figure out how to store data so it can be retrieved later. Not to worry, enigma has made it very easy to store data without worrying about relational data too much.

We need to use a driver to talk to our database. Don't worry, Enigma takes care of this for you. But you will need to be aware that we are using it to do operations on the database.

- [Motor Documentation](#)
- [PyMongo Documentation](#)

Let's begin by adding an add sub command to our existing command to allow adding new tags. In order to do this, we need to use a feature of the `discord.py` library called Groups. You can learn more about groups [here](#).

```

class Tag:
    def __init__(self, bot):
        self.bot = bot
        self.data = plugin_data

        # self.bot.db.database is simply the name of the database
        # you provided in the app's config file.
        # self.db is a Database object much similar to what's found in
        # PyMongo. Except, here we are using the asynchronous version
        # of the library called Motor.
        self.db = self.bot.db[self.bot.db.database]
        self.logger = self.bot.logger

        # Notice this is now group()
        # Setting invoke_without_command=True makes sure sub commands don't
        # run the code in this function when they are called.
    @commands.group(

```

(continues on next page)

(continued from previous page)

```

        name="tag",
        invoke_without_command=True
    )
    @commands.guild_only()
    async def tag(self, ctx, tag: str):
        document = await self.db.tags.find_one({"tag": tag})
        self.logger.debug(f"Tag: {tag} | Document: {document}")

    # Notice that we are using the tag coroutine as a decorator here.
    @tag.command(name="add")
    async def add_tag(self, ctx, tag: str, *, content: commands.clean_content):
        """
        Notice the '*' used after the tag param. This will ensure that
        the content of the message after the tag won't get passed into
        our coroutine. In short, without the '*', it will raise an
        error for more than one argument after the tag.

        With the '*' it will consider everything after the tag as a
        string with newlines and spaces intact. commands.clean_content
        also makes sure the input is more clean and will do some
        parsing for you.
        """

        # Let's insert our first document into the collection.
        # MongoDB is lazy when creating collections. It is a convention
        # to name collections after the cog or the extension to make it
        # easier to locate. Here this line will create a tags
        # collection as well as insert the json file as a document.
        self.db.tags.insert_one(
            {"guild_id": ctx.guild.id, "tag": tag, "content": content}
        )

```

Run `+tag add mytag 123` or something similar (preferably with newlines and spaces as well) from discord to ensure there are no errors. Then check you MongoDB client to make sure that the document's were inserted.

If all went well we can add some code to display the tags.

```

@commands.group(
    name="tag",
    invoke_without_command=True
)
@commands.guild_only()
async def tag(self, ctx, tag: str):
    # Find the document with the tag that we inserted earlier
    document = await self.db.tags.find_one({"tag": tag})
    self.logger.debug(f"Tag: {tag} | Document: {document}")
    if document:
        # Send a message to the guild with the content
        await ctx.send(document["content"])
    else:
        # These are embeds that make thinks look prettier. Here we
        # made a simple error message.
        response = discord.Embed(
            color=0x7F8C8D,
            title=" Tag does not exist! "
        )
        await ctx.send(embed=response)

```

(continues on next page)

(continued from previous page)

```

@tag.command(name="add")
async def add_tag(self, ctx, tag: str, *, content: commands.clean_content):
    # Let's check to make sure the tag doesn't already exist.
    document = await self.db.tags.find_one({"tag": tag})
    if document:
        response = discord.Embed(
            color=0x7F8C8D,
            title=" Tag already exists! "
        )
        await ctx.send(embed=response)
    else:
        self.db.tags.insert_one(
            {"guild_id": ctx.guild.id, "tag": tag, "content": content}
        )

```

That wasn't too hard was it? Let's add some more commands and functionality to make a full blown plugin. You can see the full code here.

```

import discord
from discord.ext import commands

plugin_data = {
    "name": "Tags"
}

class Tag:
    def __init__(self, bot):
        self.bot = bot
        self.data = plugin_data

        # Easier Access
        self.db = self.bot.db[self.bot.db.database]
        self.logger = self.bot.logger

    @commands.group(
        name="tag",
        invoke_without_command=True
    )
    @commands.guild_only()
    async def tag(self, ctx, tag: str):
        document = await self.db.tags.find_one({"tag": tag})
        self.logger.debug(f"Tag: {tag} | Document: {document}")
        if document:
            await ctx.send(document["content"])
        else:
            response = discord.Embed(
                color=0x7F8C8D,
                title=" Tag does not exist! "
            )
            await ctx.send(embed=response)

    @tag.command(name="add")
    async def add_tag(self, ctx, tag: str, *, content: commands.clean_content):
        document = await self.db.tags.find_one({"tag": tag})

```

(continues on next page)

```

    if document:
        response = discord.Embed(
            color=0x7F8C8D,
            title=" Tag already exists! "
        )
        await ctx.send(embed=response)
    else:
        self.db.tags.insert_one(
            {"guild_id": ctx.guild.id, "tag": tag, "content": content}
        )

@tag.group(
    name="delete",
    invoke_without_command=True
)
async def delete_tag(self, ctx, tag: str):
    document = await self.db.tags.find_one({"tag": tag})
    if document:
        await self.db.tags.delete_one({"tag": tag})
    else:
        response = discord.Embed(
            color=0x7F8C8D,
            title=" Tag not found! "
        )
        await ctx.send(embed=response)

@tag.command(name="list")
async def list_tags(self, ctx):
    tags = []
    async for document in self.db.tags.find({"guild_id": ctx.guild.id}):
        tags.append(document["tag"])
    if len(tags) > 0:
        await ctx.send("\n".join(tags))
    else:
        response = discord.Embed(
            color=0x7F8C8D,
            title=" No tags to list! "
        )
        await ctx.send(embed=response)

@delete_tag.command(name="all")
async def delete_all_tags(self, ctx):
    await self.db.tags.delete_many({"guild_id": ctx.guild.id})
    response = discord.Embed(
        color=0x7F8C8D,
        title=" All tags deleted! "
    )
    await ctx.send(embed=response)

def setup(bot):
    bot.add_cog(Tag(bot))

```

Congratulations! You reached the end of this tutorial. You should now have sufficient knowledge to make more kinds of plugins.

Advanced Tutorials

1.7 enigma

1.7.1 enigma package

Subpackages

`enigma.commands` package

Submodules

`enigma.commands.start` module

Module contents

Submodules

`enigma.app` module

`enigma.client` module

`enigma.utils` module

`enigma.utils.find_members` (*ctx*)

Parses arguments passed to a command and returns a list of me

Parameters `ctx` – pass a `discord.ext.commands.Context` object

Returns a list of `discord.Member` objects

`enigma.utils.get_command_args` (*ctx*, *lower_case=True*)

Gets the arguments passed to a command.

Parameters

- `ctx` – pass a `discord.ext.commands.Context` object
- `lower_case` – returns arguments in lower case

Returns `list`

Module contents

CHAPTER 2

Indices and tables

- `genindex`
- `modindex`
- `search`

e

`enigma`, 15

`enigma.utils`, 15

E

enigma (module), 15
enigma.utils (module), 15

F

find_members() (in module enigma.utils), 15

G

get_command_args() (in module enigma.utils), 15