

The University of Akron

IdeaExchange@UAkron

Williams Honors College, Honors Research
Projects

The Dr. Gary B. and Pamela S. Williams Honors
College

Spring 2023

Discord API Wrapper

Joshua Brown
jgb38@uakron.edu

Follow this and additional works at: https://ideaexchange.uakron.edu/honors_research_projects



Part of the [Other Computer Sciences Commons](#)

Please take a moment to share how this work helps you [through this survey](#). Your feedback will be important as we plan further development of our repository.

Recommended Citation

Brown, Joshua, "Discord API Wrapper" (2023). *Williams Honors College, Honors Research Projects*. 1675.

https://ideaexchange.uakron.edu/honors_research_projects/1675

This Dissertation/Thesis is brought to you for free and open access by The Dr. Gary B. and Pamela S. Williams Honors College at IdeaExchange@UAkron, the institutional repository of The University of Akron in Akron, Ohio, USA. It has been accepted for inclusion in Williams Honors College, Honors Research Projects by an authorized administrator of IdeaExchange@UAkron. For more information, please contact mjon@uakron.edu, uapress@uakron.edu.

Discordwrap, a Python Library

By

Joshua Brown

Honors Project

Department of Computer Science

Spring 2023

Edited by

Matthew Dray

Table of Contents

Abstract	3
Preface	3
Part 1. Use Cases	4
Part 2. Design	5
2.1 Rate Limiting	5
2.2 Async vs Sync	6
2.3 Authentication	6
2.4 Errors	7
2.4 Methods	7
2.5 Testing	7
2.6 Black	8
2.7 Just	9
2.8 Continuous Improvement and Continuous Development	9
2.9 Implementation	11
2.10 Challenges	12
Part 3. Public Docs	14
3.1 Design	14
3.2 CICD	14
Part 4. Reflection	15
4.1 Mistakes	15
4.2 What Was Learned	16
4.3 Future Work	16
Part 5. Links to Resources	16
5.1 Discordwrap	17
5.2 Brunus Labs	17
5.3 Conventional commits	17
5.4 Pytest	17
5.5 Poetry	18
5.6 Just	18
5.7 Black	18
5.8 Nextcord & Discord.py	18
5.9 Nextra	19
5.10 Vercel	19
5.11 Discord	19

Abstract

Discordwrap is a Python library that abstracts the Discord API so that developers can easily integrate their existing projects with Discord. This paper outlines Discordwrap's creation, from start to finish, including implementation as well as key design decisions, such as the decision to provide a functional library interface rather than an object-oriented one. There are several reasons for this decision, the most pertinent being the stateless nature of a functional library removes the need to pass around a class object like a “client”. On top of that, many object-oriented and bot-based Python libraries for Discord already exist, none of whose goal is to provide a simple, yet effective functional API wrapper in a synchronous manner, such as that of Discordwrap. The goal of this paper is to both outline and document the key design decisions and implementation details of the library through a retrospective lens.

Preface

Discord as an application creates an amazing ecosystem designed to bring individuals together to form communities, share experiences, play games together, host study sessions, and communicate through virtual means. This amazing ecosystem is possible due to the developer-empowering API Discord has made public which encourages developers to create bots and integrate other services into the ecosystem.

Throughout my research project, the developer community has re-establish a solid variety of well-maintained Python libraries including nextcord, Discord.py, interactions-py, etc. These libraries all share a common goal of wrapping Discord bot interactions for bot developers, meaning they focus on a single bot instance and how it handles various user and Discord

interactions. This is what sparked the motivation for Discardwrap: create a class-less, rate-limit aware, simple API wrapper for Discord. Even in its simplest form, Discardwrap has proven useful and is already integrated into multiple projects under development by Brunus Labs¹.

Part 1. Use Cases

The most immediate use case for Discardwrap is for sending notifications to Discord without the need for a bot or webhooks. For example, in one of Brunus Labs's projects, a client wanted Discord notifications when a given event was triggered on their website. This could be solved by webhooks, however, webhooks have a much less forgiving rate limit policy than bot messages, and each webhook must be explicitly set up. Messages from bots on the other hand have more freedom and have a much higher rate limit threshold. This is where Discardwrap comes into play, by handling the rate limiting, and message sending to the Discord API from the website's flask² backend, and not from a user. This also saves the project from needing a separate application just for the bot.

Another client of Brunus Labs needed internet data to be aggregated and sent to their Discord server, with the messages being controlled by multiple cron jobs. This is another example of where a Discord bot fails to solve the problem, as the messages are being initiated by Python scripts, and run till completion. Discardwrap can elegantly handle this situation, requiring one line of code for setting up the token, and one line of code for sending the actual message.

¹ Brunus Labs is a company founded by the author, Josh Brown. Josh started the company in September 2022 in his last year of college, taking on freelance-like contracts under the name of an official LLC.

² Flask is a backend framework for python, used to create APIs or full web apps.

Part 2. Design

At its core, Discordwrap is meant to be an extremely lightweight library with minimal overhead. It has zero dependencies and only abstracts what it needs for the end user. There are no classes for messages, users, channels, etc, and the goal is to provide no additional documentation overhead to what Discord provides. In keeping with this philosophy, Discordwrap's methods directly return the JSON data from the Discord API.

2.1 Rate Limiting

Discordwrap's main functionality is taking care of the complex rate limiting that Discord strictly enforces. In the eyes of rate limiting, Discord treats sections of its API as buckets, wherein each bucket has an independent rate limit. The problem lies in making sure that the library blocks lazily, and not prematurely. For example, if the rate limit for a given bucket is five messages, and the user initiates a sixth message to Discord, it should only block the user's program on the sixth message, and not on the fifth message when the rate limit was initially hit. This ensures that the user's program remains as fast as possible since the library only blocks when it's sure that there is a message that needs to be sent but must wait for its rate limit bucket to expire.

Internally, this is solved using a combination of multithreading and mutexes with the `asyncio` library (which is included with Python, and therefore not an external dependency). When the user requests an API call to Discord, the library creates a singleton that holds a thread to handle mutex blocking and callbacks. This singleton, by definition of the pattern, is only ever created once, and other calls to instantiate it will just return the current singleton. Once the

singleton is created, the next step is to grab the mutex for the given bucket, along with a global mutex, in case there is a rate limit for the Discord API as a whole. Once both mutexes are acquired, Discordwrap proceeds by locking a new mutex for the given bucket. Then, the API is called on the given route, checks if a global, or local rate limit has been reached, and returns the JSON to the caller. If either rate limit is hit, then it fails to unlock the mutex after the function returns and initiates a callback that unlocks the mutex after the duration of the rate limit, wherein the lock will be released on the singleton.

2.2 Async vs Sync

The second challenge this library presents is not adding the mental overhead of asynchronous code in synchronous scripts. To solve this, Discordwrap uses two function decorators. The base function itself is synchronous, since the requests library also included in Python is synchronous. The first decorator takes this function and wraps it inside an asynchronous function that takes care of rate limiting. This is done so that our rate-limiting logic can be reproduced regardless of the type of request sent to Discord (POST vs GET vs PUT, etc). Lastly, this is wrapped again in a function that takes the asynchronous function and runs it inside the singleton's main thread until it completes and then returns the result, therefore making the function synchronous again.

2.3 Authentication

To authenticate requests to Discord, the user's bearer token needs provided, which is given to them by Discord itself. This is essentially the user's API key. Rather than instantiating a

class like most libraries do, the easiest way for this library is to use a static class variable, aptly named `Auth`, and assign it the token, persisting it for the duration of the running program.

2.4 Errors

Most errors that can be thrown from Discord are also implemented in this library. Errors are thrown based on responses from Discord or can be caught before a request is generated, depending on the required syntax of the different library methods. For instance, in order to send a message via the `create_message` function, a user must supply one of `content`, `embeds`, `sticker_ids`, or `components` as a key in the JSON body. If they fail to do so, the library will throw an `InvalidBody` error before the request is ever attempted.

2.4 Methods

Once the core logic for rate limiting was implemented, writing out actual Discord methods became straightforward. The input parameters lined up with the required fields for the API URL, such as `channel id` or `guild id`³ along with the JSON body needed for the request. The request then returns the JSON response from Discord or the status of the response if no request body is returned.

2.5 Testing

Any good library needs testing. For `Discordwrap`, the testing library was chosen to be `pytest`, for its robustness and simplicity. Tests are written in the `tests` folder of the project, and are organized into files that are prefixed with `“test_”`. Since libraries are often worked on and

³ Guilds in Discord are what most people refer to as servers. In the Discord API however, they are explicitly referred to as Guilds.

improved, tests must cover as much as they can to ensure nothing breaks as the developer changes source code. Since this library is wrapping an API, requests to the Discord API shouldn't be made every time tests are run. To solve this, we take advantage of monkey patching, a method whereby the API caller function is swapped with a mock function written inside the library that returns fake JSON from a previous response. In other words, once a new function is written, a developer can call it on the Discord API, and save the response in a JSON file that can then be injected into the caller functions used by tests.

Of course, this process is automated. Contributors and developers can use the `get_payload.py` script that prompts the user for the endpoint they want to call, along with the type of request and method name, and automatically save the file in the proper location with the desired JSON data, including proper formatting.

2.6 Black

When writing code, it is important to decide and implement a formatting standard for the code itself. For Discordwrap, black was chosen as the code formatter, as it is extremely popular in the Python community. When integrated properly with a developer's IDE, black will automatically format code when a file is saved so that the developer does not need to spend time fixing formatting errors manually. Black implements various rules such as 2 spaces between functions, no parentheses on "if" statements, max line width, and so on. This ensures that the code follows a consistent style no matter what developer is working on the project, and keeps the code clean.

2.7 Just

Any project is not complete without good automation or tooling. ‘Just’ is one of the best ways to simplify a developer’s tooling on the CLI, where many developers do most of their day-to-day tasks. ‘Just’ is essentially a Makefile but for CLI commands. Recipes can be created that will call CLI commands in order. Recipes can depend on others, have variables, and have other useful features. Without ‘Just,’ running tests would require the developer to build and install the library, followed by running black and pytest. These 4 commands all have unique and different syntaxes. ‘Just’ simplifies these by combining them in the justfile to ‘just test’. Testing is not the only thing ‘Just’ is used for. In fact, all CLI commands that are used more than once specifically for the project are added to the justfile.

2.8 Continuous Improvement and Continuous Development

The last piece of this project is CICD. Projects that have the potential to be developed by more than one person, such as open source projects, need good CICD to ensure that all of the best practices are being followed, and to increase developer productivity.

To understand Discordwrap’s CICD pipeline it’s important to first discuss how changes are introduced via git. First, an issue is made on Github, describing what new features need to be made or bugs need to be fixed. Then, a developer creates a branch following the pattern “[feat|fix]/[issue number]-[short-description]”. From here, they work on their branch, complete the necessary changes, and then create a pull request into the main branch. This is when all of our CD pipelines run. Once the PR is complete, our releaser runs, continuously updating the library after every change. This lets developers update Discordwrap very quickly. In one instance, a hotfix for a bug was able to be released to PyPi in under 5 minutes after being found.

Discordwrap uses an impressive level of CICD, broken into multiple sections. The first section is meta linting, followed by code testing, and lastly, code releasing. All workflows can be found in the `.github` folder of the project, using GitHub actions as our CICD runners.

Meta linting is the process of linting commit messages, PR titles, and branch names for pull requests into the main branch. This ensures that all developers are following a strict standard for commit history and pull request formations. These are specifically important for the releaser workflow. This is triggered on every pull request, regardless of what branch is being merged into.

Next, also run on every PR, is code testing. This involves running the pytest functions, along with black, making sure that the code is working properly and formatted correctly. If any of these tests fail, the requesting developer must fix the issues in further commits until all tests pass.

The last workflow, and by far the most impressive, is the releaser. Discordwrap uses a commit standard known as Conventional Commits, where each commit follows a strict format that includes the type of change, a short description, and an optional location. Since all of the repository's commits strictly follow this standard, it's possible to automatically generate changelogs based on the commits in a given PR. It can also automatically bump the version number of the project based on semantic versioning, where each number in the version of a project denotes a specific type of change in the project's code. When the workflow runs, it appends the new changes from the merged PR into the running changelog, bumps the version number, and then automatically uploads and updates the library code and version to the PyPI repository. In other words, the deployment process is 100% automated, with no developer input needed, other than the signoff and merging of a pull request.

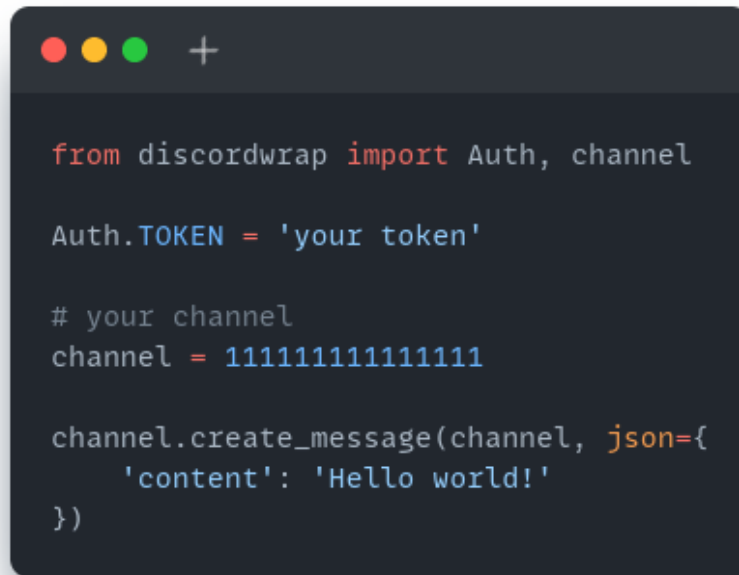
There are also two other pieces of CICD, although they are much more minor. First is the simple pull request template, which gives developers a soft template for pull requests into the library. Second is the dependabot workflow. This is a bot that runs once every day and checks the versions of GitHub workflow libraries and ensures they are always up to date.

2.9 Implementation

The technical development environment for Discordwrap falls into two categories. First, is the development environment, where maintainers and contributors may add features or fix bugs within the library. The second is the end-user environment, where users implement Discordwrap into their projects.

To make this as streamlined as possible, Poetry was used to ensure every developer's environment was the same when working on the library. Poetry describes itself as an easy Python dependency management tool, where it can take care of not only building the library, publishing to PyPi, and installing developer dependencies all via one simple file, with a few simple commands. Development was primarily done on Linux, however, Poetry is OS agnostic and Discordwrap therefore can be worked on with any operating system once installed.

The second category is that of library implementation. To use the library, the user must first set their API token, given to them by Discord. Once the token is set, the user can use any method that is implemented in Discordwrap in the codebase. As shown in the figure, the library gives the user full control over how they want to send the request with direct access to the JSON sent to the API. The result from this method will also be the JSON result from the API itself. The user, therefore, does not need to depend on nor learn documentation specifically for Discordwrap, but only needs to rely on the Discord Developer documentation.



```
from discordwrap import Auth, channel

Auth.TOKEN = 'your token'

# your channel
channel = 1111111111111111

channel.create_message(channel, json={
    'content': 'Hello world!'
})
```

2.10 Challenges

Creating a Python library is not an easy task. Learning how to properly structure the project, learning how imports worked, and how packages are built was very challenging. The best advice here to recommend to others for overcoming similar challenges is to go to the documentation first and pair that with examples that can be found in the real world. For Discordwrap, that looked like comparing and examining other open-source libraries, along with reading the documentation for creating libraries directly from Python's documentation.

The second challenge that was encountered was properly setting up our CICD pipelines. There is a lot of infrastructure in this simple library, as mentioned previously. The most challenging of those was the continuous release. To automatically release the library on a successful PR, Discordwrap's pipelines must not only determine the new version number automatically from the commits within the new PR but also make a commit in the pipeline itself, bumping the version number everywhere it needs to be bumped. To make it clear, Discordwrap's

CICD automatically commits code changes triggered by pull requests. This was a great accomplishment once it started working, as this is the main step that lets the library automatically release to PyPi, the Python package repository.

With the framework of the library laid out, and CICD set up for easy releases, the next challenge was to create a proper testing framework for the library. As mentioned, pytest was the chosen testing framework, as its performance in prior projects proved acceptable for this one. The trick was not in the testing framework itself, but in making tests that were able to mock Discord requests so that tests could be run without actually hitting Discord's API. The solution here was one that was learned about before, and even used, but not to this extent or capacity. Monkey Patching the requests library and mocking the API requests from the JSON file for the given endpoint that had already been generated was a great feat and made testing from then on a breeze. New tests now only need to be run one time on the actual API and thereafter could be faked from then on.

Now, what remained was the core library. Here lies the greatest challenge of the project and the core of this research paper; Writing a functional Python library for Discord. As mentioned before, this was indeed solved and proven possible, thanks to a singleton that holds a reference to the current thread dedicated to the Discordwrap library, with some mutex logic stopping requests from being sent once the library exceeded its rate limit, though only for a given bucket.

When first building the rate portion of the library, the mutexes simply did not work properly, either refusing to unlock or not running synchronously after completion. After a few weeks of research and digging into StackOverflow questions along with the asyncio library documentation, the solution finally became clear. The Singleton, on the first request, starts a

thread dedicated to the Discordwrap library, that the mutexes live on. When a rate limit is hit, a mutex release is thrown on a callback on the thread inside the singleton, so that it can return to the user's main program as fast as possible after sending a request. The async problem was solved by wrapping the async functions in decorators that run the async function to completion, making it artificially synchronous.

Part 3. Public Docs

3.1 Design

The documentation (docs) are provided using a simple static site built with Nextra. Nextra makes developing documentation as simple as adding markdown-x or markdown files to a git repository. This means that developing code blocks and formatting text is very simple. Pages are automatically created and linked together, along with links to the project GitHub.

The site is broken up into a few pages. Firstly, the Introduction page, lays out the reasons behind Discordwrap's existence, its use cases, and its features. Next is the Quickstart page, which has instructions for getting started with the library. An about page can be found on the top navigation that details some background about the project. Finally, the API Reference page outlines each method in the library.

3.2 CICD

The CICD for the documentation site is equally as impressive as the CICD for the library itself. This is thanks to Vercel's advanced infrastructure. When the main branch is updated, Vercel will instantly download, build, and redeploy the code to production. This entire process

generally happens in under 1-minute after a push. If a build fails, Vercel will roll back to the last successful deployment to keep the site alive. All of this is within Vercel's free tier, with the only cost being the domain name of the site.

Part 4. Reflection

Building a library is hard. There are a lot of moving pieces and very tough designs that go into laying the foundation for a good library, regardless of purpose. It was also difficult to determine what the project needs or should need before the first line of code is written. Originally, Discordwrap was to have a site, documentation, and more. These quickly got pushed to the side as work was focused on developing state-of-the-art CICD, developer tooling, testing, and the library itself.

4.1 Mistakes

Getting the core rate limiting correct was a pain. Python's handling of async threads is subpar, at best. Thankfully, nextcord's implementation which was forked from Discord.py, had the foundations of rate limiting that provided a reference for this library, translating it to a more functional approach.

That's what programming is though. Mistakes will be made, sometimes over and over, until the right solution is found. Mistakes are only mistakes until fixed and learned from.

4.2 What Was Learned

Dealing with asynchronous code, and getting a solid grasp on multithreading in Python are easily the greatest and most important topics learned during this project. Also, the importance and weight a good foundation plays in building a library. This is not the first library I have developed, however, it is the first one I am happy with. It not only works but works well and can be used in production code.

Another interesting topic I was able to dive into was learning Poetry, and how it handles library development. I feel much more confident in designing and working with library fundamentals and virtual environments thanks to poetry.

4.3 Future Work

Discordwrap still needs a lot of love. The most immediate tasks are making a homepage, explaining its purpose and how to get started, along with implementing more Discord endpoints. More advertising also needs to be done within the Discord developer community so that people might start using the library in the wild. There are also some client projects that can immediately benefit from this library that should be bumped in priority..

Part 5. Links to Resources

This project used a multitude of tools, libraries, and standards from various sources that have been named multiple times above. This list details what was used, along with links to each project.

5.1 Discordwrap

Discordwrap can be found at the Brunus Labs Github at <https://github.com/Brunus-Labs/Discordwrap> and can also be installed via pip with “pip install Discordwrap”. The package can also be seen at <https://pypi.org/project/discordwrap/>. The public docs are hosted at <https://discordwrap.brunuslabs.com>.

5.2 Brunus Labs

Brunus Labs is still, as of writing, very early in the startup stage. We are currently focusing on our current clients, and our main website is still under development. Feel free to check it out periodically for updates. You can find us at <https://brunuslabs.com/> and on our GitHub at <https://github.com/Brunus-Labs>.

5.3 Conventional commits

Conventional commits is a popular standard for commit messages so they can be used in automatically generated change logs and version bumping. More information can be found at <https://www.conventionalcommits.org/en/v1.0.0/>.

5.4 Pytest

Pytest is the testing library for Discordwrap. More information can be found at <https://docs.pytest.org/en/7.2.x/>, and it can be installed with “pip install pytest”.

5.5 Poetry

Poetry is the backbone of the library structure, helping with developer dependency management, and library building. Their homepage can be found at <https://python-poetry.org/>.

5.6 Just

Just is an extremely powerful CLI runner that has seen usage in a multitude of personal and professional projects, no matter the size, and deserves more attention and use. Their main page can be found at <https://just.systems/> where you can also find their GitHub and manual pages.

5.7 Black

Blacks is the name of the formatter. Black specifically advertises itself as a non-configurable formatter, where anyone using black is bound by the same rules and format. Their homepage can be found at <https://black.readthedocs.io/en/stable/>.

5.8 Nextcord & Discord.py

Nextcord and Discord.py helped greatly in not only inspiration but also the rate-limiting part of the library. Their docs can be found at <https://docs.nextcord.dev/en/stable/> and <https://Discordpy.readthedocs.io/en/stable/> respectively.

5.9 Nextra

Nextra is a static documentation / blog template built with nextjs by vercel. Nextra can be found at <https://nextra.site/>

5.10 Vercel

Vercel is the host for the Discordwrap docs, which also took care of the CICD for the site. Vercel can be found at <https://vercel.com>

5.11 Discord

The Discord docs for developers are a great way to get started with working with APIs and are what sparked the interest in the space. Their API docs can be found at <https://Discord.com/developers/docs/intro> and Discord's homepage can be found at <https://Discord.com/>