The GUPHY – GIF Machine

Inhalt

[Layout 2](#_Toc161413799)

[Libraries 2](#_Toc161413800)

[Methodology 2](#_Toc161413801)

[Structuring 2](#_Toc161413802)

[Restructuring 3](#_Toc161413803)

[Error Handling and Debugging 5](#_Toc161413812)

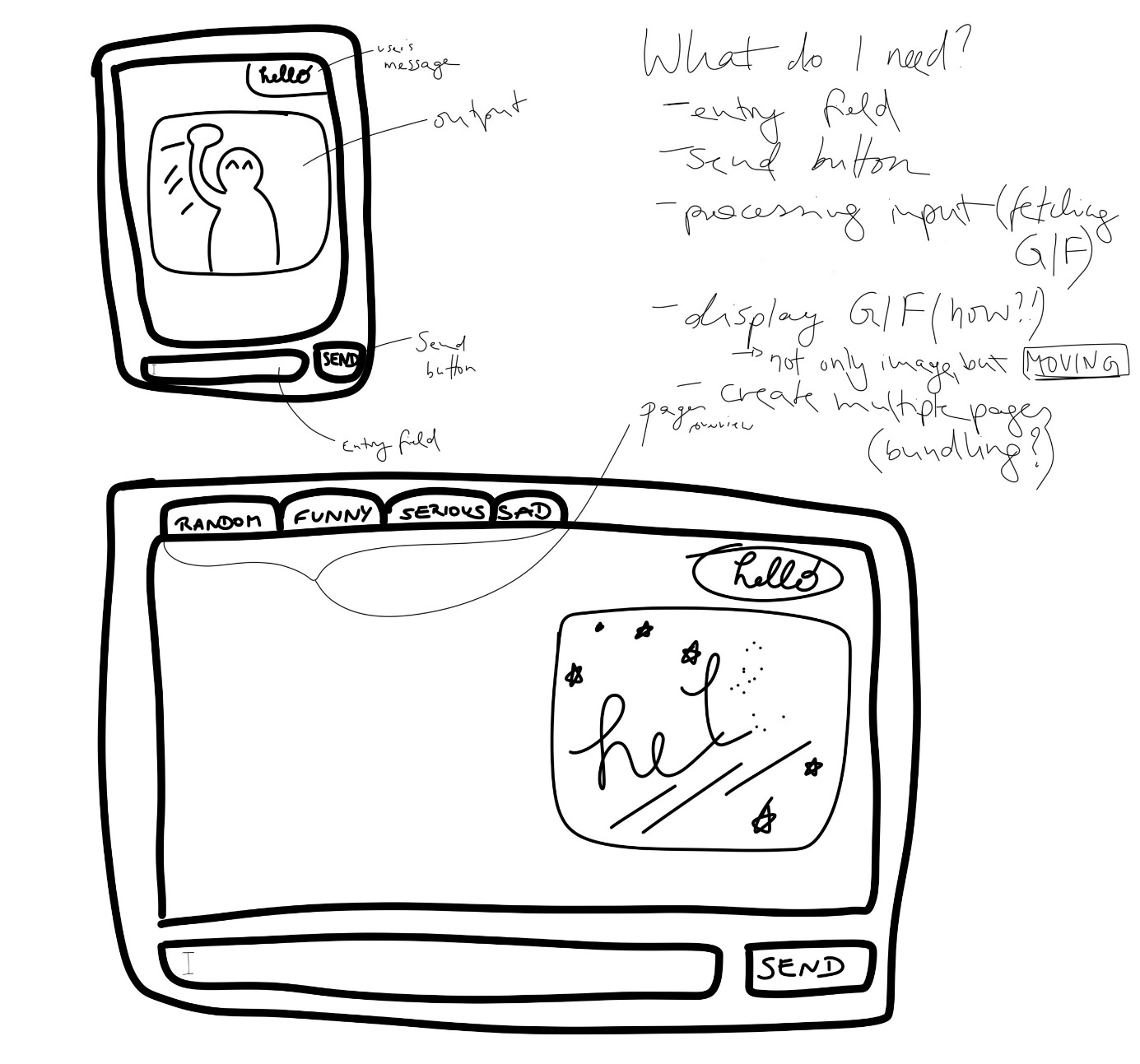
[Limitations 5](#_Toc161413813)

Layout  
As I am a professional in first impressions; I will introduce my weakest link first. Maybe so we can move on to the more fun stuff. Maybe so it will only get better from here on out. I have to be honest, I could have spent more time on making the application prettier. It is very basic and I wish I could have made it look a little more modern. However, I wanted to use a clean layout that resembles a classic chat design so I focused on that for a start. It isn’t very special, but it’s easy to navigate. To be honest, I could have spent more time here. The thing that bugs me most about the layout is that the messages are only stacked underneath one another if not expanded. And then, the messages are sometimes cut off. I started playing around with a dark mode, using a toggle definition, but I haven’t gotten it to work in time. In my repo, you’ll find a first draft I was still trying to integrate at the moment of submitting. However, my script does run and is in pretty ok shape to present, so here goes the rest of the report.

Libraries  
As we were working with tkinter and PIL, these are the basic libraries I worked with. I then integrated requests to work with the API request. For the API retrieval, I first implemented the webbrowser module to open the GIFs. However, I wanted the GUI to be usable without opening a web browser for each message. So, I implemented BytesIO in order to manipulate the images and then display them directly in my application without saving them permanently. The application was running very slowly from time to time, so using a binary stream helped speed things up greatly.

# Methodology

Structuring  
For the basic functionalities, I used a simple structure. In the very first version, I only included the 3 following functions: send\_message, display\_message, process\_user\_input. For simplicity, the script only mirrored the user’s input first. From there, I thought about how I could implement more features. However, with the complexity of what I wanted to include, I knew I would run into problems quickly. So, I restructured quickly, bundling functions into bigger bits.



A first draft of my design

## Restructuring

I quickly ran into bugs and limitations writing my code from top to bottom with a single block of code. It became bulky and hard to maintain an overview. So instead of endlessly re-reading this huge ugly block of code, I restructured to find out how I could best bundle the script into main functions. In order for my script to run, I determined 5:

1. Create a page (ln 14-40)

I quickly restructured the code to create a frame for the pages so I could model them all the same way (create\_page()). I then only had to create a single function for all three pages, including the main features of the application. And it is highly modular, making it easy to add on more pages as desired as well as change the layout for all three of them.

2. Display the user’s message (ln 43-68)

Here, I used the message widget to organize the written messages into a frame in the chat window, displaying first the user’s messages and then, with a short delay, the GIFs. This function includes a user\_label as well as a gif\_label. The phrase “GIF will be displayed here” proved very useful with some impatient test runners.

3. Process the user’s input (ln 72-90)

A simple function to process the user’s input (for more see docstring).

4. Fetch a GIF from the endpoint (ln 94-140)

Now, this one was tricky. I’m fairly new to working with APIs. For example, I didn’t know to use the isinstance() function. I only knew to use the requests library, but not how exactly so, I asked ChatGPT for advice on how to write the function. Ln. 114-127 are not my own but taken from ChatGPT. In the first block, the function sets up the parameters for the API request. It includes the GIPHY API key in the params dictionary. Then sending a GET request to the API, checking for any HTTP errors. So far, so good. From ln 181-189, the function checks if the response contains a "data" key, and if the "data" is a non-empty dictionary. If so, it attempts to extract the original image URL from the response. If the URL is found, it is returned. If not, it prints an error message.

5. Display the fetched GIF (ln.142 -195)

As it turns out, the last step I had to divide into 2 as displaying the GIF in a sequence of images was its own challenge.

5.1 Displaying the GIF label (ln. 142-170)

Most definitely the easier of the two functions. If a valid GIF URL is provided, the function uses the requests-get method to make a GET request to the provided gif.url and retrieves the content. The data is being processed; read and stored in a BytesIO object named gif\_data. This object is then used to create an Image Object from PIL using the Image.open() method. I then had to convert the object into a Tkinter-compatible object (ImageTk.PhotoImage). By now, I had an image in the application when I ran it. However, it was missing a vital piece: the moving of the images!

### 5.2 Displaying a moving image (ln. 173-195)

For getting the image to actually move and not be a still picture, I had to get creative. As I was looking into different ways to work with GIFs in python (GIFImagePlugIn being one of the more promising ones), I always ran into the same problem: performance. The GIFs would lag greatly, making me seek another solution. With the Pillow library and the seek() method, the script is now able to navigate through the frame sequences by going through each subsequent frame one by one.

### 6. Launching the app

In this very last bit, I launch the GUI and create the pages I want using either my general endpoint or specific search endpoints using the search tags.

# Error Handling and Debugging

Before I even started writing the current version of the script, I looked at examples of modular coding so I could bundle my functions as much as possible. It was a challenge integrating my list-style form of functions into this design, but definitely worth it.   
A great portion of my time went into rewriting and debugging faulty parts of my code. Spelling mistakes and indentation errors were among the most of them, but of course, some bigger problems were also playing a role. I included several try-except blocks which came in especially handy when trying to understand the API connection (even working with ChatGPT, it didn’t work out right away but took trial and error).   
I have also tried out different endpoints and experimented with different scenarios. However, in the end, sticking to these basic principles and always coming back to my main structure provided the cleanest experience. I have to say the Restructure of my final submission was not the second try. It took a lot of time and consideration to get here, but in the end, it was always worth it to look at the bigger picture when I was stuck and regroup.

# Limitations

I wanted to integrate the user’s message as possible tags and have the script respond with fitting results. I tried to implement the user’s messages to be read as a tag and then retrieve a more fitting GIF from the random API endpoint. However, as time ran out, I had to give up this idea of creating more tailored answers by the bot. The returned GIFs from the search results (funny, serious, and angry) also seem to often come from the few first search results for the category, leading to an ultimately monotonous experience if used over a longer period of time. I am confident that implementing this idea would make the whole app a lot more playful and engageable. As March 15th is coming closer, I am currently working on a dark mode you can switch to so the design is a bit more rounded. In my repo, you will find a snippet called “toggle.py” where you can see my process. It is in no shape to be integrated into the script yet, but I am hopeful I will get it to run in the future.