# Frizz-Wiz Application Documentation

## Overview

Frizz-Wiz is a web application running on an Nginx server. The application is configured to be accessible from the internet via the domain names **frizz-wiz.com** and **www.frizz-wiz.com**.

## The Application Code

### The Components

1. **Python Scripts**: Various Python scripts are used in the Frizz Wiz application to handle different tasks. These scripts include functions for retrieving weather data, geolocation information, and slider-based time selection. They interact with external APIs, such as OpenWeatherMap to fetch weather data and geolocation details. The scripts are called by the Flask application to provide the necessary data for rendering the web pages.
2. **HTML Templates**: The HTML templates define the structure and content of the web pages displayed to users. They utilize template engines, such as Jinja2, to dynamically insert data retrieved from the Python scripts. The templates include forms for user input, sliders for time selection, display weather results, and render appropriate images based on weather conditions.
3. **JavaScript**: Multiple JavaScript files are used to handle the UI interactivity, geolocation functionality, and to make AJAX requests to the server. These include the **new\_slider.js** for time selection via sliders, **fetch\_forecast.js** to fetch weather data based on slider selection, and **location.js** to get current latitude and longitude using browser's GPS.
4. **Gunicorn Configuration and Error Logs**: Gunicorn is a Python WSGI HTTP Server that serves the Flask application. The configuration file specifies server settings such as the number of worker processes, bind address, and log file locations. The error logs capture any errors or exceptions that occur during the server's operation, providing valuable information for debugging and troubleshooting.
5. **Flask Application**: The Flask application is the core of the Frizz Wiz web server. It handles incoming HTTP requests and generates appropriate responses. The application consists of routes that define the URL endpoints and associated functions to handle the requests. It also utilizes templates to render dynamic HTML pages and JavaScript files for the interactivity of the pages. The application interacts with the Gunicorn server to serve the web pages to clients.
6. **Gunicorn Server**: Gunicorn is responsible for running the Flask application as a web server. It listens for incoming requests and dispatches them to the appropriate Flask route for processing. The server configuration determines how the server operates, including the number of worker processes, concurrency model, and network settings.
7. **Waitress Server**: Waitress is an alternative WSGI server to Gunicorn. It can be used to run the Flask application instead of Gunicorn. It provides similar functionality but with different configuration options. The Waitress server listens for incoming requests and routes them to the Flask application.

By combining these components, the Frizz Wiz application functions as a web server that receives user input, retrieves weather data, and dynamically renders HTML pages with the corresponding information. The server (Gunicorn or Waitress) listens for incoming requests and directs them to the Flask application, which utilizes Python scripts and HTML templates to process the requests and generate the appropriate responses. The end result is a user-friendly interface where users can retrieve weather information based on zip codes or their current location.

web\_app.py

Flask App

frizz\_landing.html

User

web\_main.py

Weather Data

run\_waitress.py

(local web server)

Fetch\_forecast.js – time select

location.js – geolocate

update\_hour\_display – hours

new\_slider.js – slider

style.css - format

APIs

Requirements.txt - Libraries Required

1. **web\_app.py**: This Python file serves as the entry point for the Flask application. It imports the necessary modules, sets up the Flask app, defines the routes, and handles the logic for retrieving weather data based on zip code or location. The routes interact with the HTML templates by rendering them and passing data from the Python scripts to be displayed dynamically. Additionally, it interacts with JavaScript files by receiving data via AJAX requests and triggering updates on the web page.
2. **weather\_fetch.py**: This Python file contains functions for fetching weather data from external APIs based on zip codes, location coordinates, or time selection. The **get\_weather\_by\_coordinates()** function retrieves weather data using the OpenWeatherMap API and performs calculations to determine the weather forecast. The **get\_weather\_by\_zip()** function does the same but uses zip codes to convert to latitude and longitude coordinates. These functions return the weather result and photo URL to be displayed in the HTML templates.
3. **frizz\_landing.html**: This HTML template is the main page of the Frizz Wiz application. It includes a form where users can enter a zip code or click a button to use geolocation. The form also includes a slider for users to select a time range for the weather forecast. It submits a POST request to the **/** route in the Flask application. It also includes placeholders for displaying the weather result, photo, and dynamically generated colors. The template interacts with the Python script **web\_app.py** by receiving the weather result and photo data to be rendered in the appropriate sections.
4. **new\_slider.js**: This JavaScript file is responsible for creating and handling the functionality of a slider that allows users to select a time range for the weather forecast. It interacts with the HTML templates to dynamically update the displayed time range based on user interaction with the slider.
5. **fetch\_forecast.js**: This JavaScript file sends the chosen time range and location details (either GPS coordinates or zip code) to the Flask app via an AJAX POST request. Upon receiving a successful response, it updates the weather forecast and the photo on the page. It interacts with both **web\_app.py** and **web\_main.py** to fetch and display updated weather forecasts.
6. **location.js**: This JavaScript file is responsible for handling geolocation functionality in the web page. It checks if the browser supports geolocation and, if so, retrieves the current latitude and longitude coordinates. It then sends this data to the server using an AJAX request to the **/location** route. Upon receiving a successful response (status code 200), it triggers a fetch for updated weather information based on the current location. This file interacts with **web\_app.py** and **fetch\_forecast.py** through the AJAX request and response handling.
7. **update\_hour\_display.js**: This JavaScript file handles the real-time update of the hour display in the Frizz Wiz application. The **updateHourDisplay** function, called upon page load and whenever the start or end hour values are changed, fetches these values, transforms them into date objects, and then displays them. This setup ensures an interactive and responsive user experience.
8. **style.css**: This is a stylesheet that dictates the aesthetic and layout for the Frizz Wiz web app. It defines styles for various elements, such as the body and range slider, to ensure an intuitive and visually appealing user interface. Customizations include font family, background, and flex container setup for the entire application, specific paragraph styles within the time-range div, and styles for the range slider's bar, handle, and layout dimensions.

These files work together to create a seamless user experience. The Python scripts fetch weather data and perform calculations, the JavaScript files handle geolocation functionality and user-selected time ranges, and the HTML template displays the content dynamically based on user input and retrieved data from the Python scripts.

### Updating Application Code

To update the application, you commit your changes to a version control system (like Git), then pull or checkout the updated code on the server.

Depending on the nature of the updates, you may need to restart the application or the Nginx server for the changes to take effect.

**Uploading Code to the Server**

You can upload code to the server using **scp** (secure copy), **rsync**, or any other file transfer method. If you use Git, you can simply **git pull** the latest version of your code on the server.

### web\_app.py or local\_web\_app.py (local test environment)

This module serves as the main entry point for a Flask-based web application, which fetches weather data based on user-inputted parameters. It supports both POST and GET requests on multiple endpoints. The **web\_app.py** file is an alternative version of the local\_web\_app.py that is designed to be run on a web server.

The **create\_app()** function initializes the Flask application, sets up CORS, and defines several routes. Notable routes include:

1. The **/** route serves the homepage, handles POST requests to fetch weather data based on coordinates or zip code, and stores fetched data in session variables. The page background color is randomly generated.
2. The **/weather\_general** route handles POST requests by delegating to a function that fetches weather data either by coordinates or zip code.
3. The **/weather** route takes POST requests and fetches weather data based on either zip code or coordinates depending on the 'weather\_option' provided in the request. It returns error messages for missing or invalid parameters.
4. The **/location** route accepts POST requests, uses received latitude and longitude to fetch weather data, and stores it in session variables. The route then redirects to the index page.

If the module is run as the main script, the Flask application is started.

Key external functions used in this module include **get\_weather\_by\_zip**, **get\_weather\_by\_coordinates**, and **get\_weather\_by\_coordinates\_or\_zip** from a module named **weather\_fetch**.

The application requires two environment variables - a "SECRET\_KEY" used by Flask for session management and potentially a "SESSION\_KEY" - although the code does not seem to use the latter. If the environment variables are not found, a fallback value is used for the "SECRET\_KEY".

In summary, **web\_app.py** defines routes and their corresponding functions for handling requests. It interacts with the **web\_main** module to retrieve weather data based on user input and renders the HTML template "frizz\_landing.html" with the results and dynamically generated colors.

web\_main.py

**weather\_fetch.py**

his script, **weather\_fetch.py**, handles fetching weather data from the OpenWeatherMap API and implements the core logic for processing the fetched data.

The script starts by loading environment variables from a **.env** file, which includes an "API\_KEY" for the OpenWeatherMap API. It also sets up the base URL for the OpenWeatherMap API.

There are three primary functions in this script:

1. **get\_weather\_by\_coordinates\_or\_zip(data)**: This function checks if the 'weather\_option' from the received data is 'zip'. If it is, it calls the **get\_weather\_by\_zip** function with the 'zip\_code', 'start\_hour', and 'end\_hour' from the data. Otherwise, it calls the **get\_weather\_by\_coordinates** function with 'lat', 'lon', 'start\_hour', and 'end\_hour' from the data.
2. **get\_weather\_by\_coordinates(lat, lon, start\_hour=10, end\_hour=15)**: This function takes latitude, longitude, start hour, and end hour as input. It defaults to 10am and 3pm if no hours are specified or if non-integer values are provided. It constructs a URL for the One Call API of OpenWeatherMap using the latitude, longitude, and API key. It then sends a GET request to this URL. If the request is successful (status code 200), it processes the returned weather data. It calculates average humidity and dew point for the specified hours and determines the forecast and advice based on these averages. It also assigns a photo URL based on the conditions. If the request is unsuccessful, it returns an error message.
3. **get\_weather\_by\_zip(zip\_code, start\_hour=9, end\_hour=15)**: This function is similar to the previous one, but it takes a zip code instead of latitude and longitude. It constructs a URL for fetching weather data based on zip code, sends a GET request to this URL, and if the request is successful, extracts latitude and longitude from the response data. It then delegates to the **get\_weather\_by\_coordinates** function to fetch and process weather data for the retrieved coordinates.

In case of a failed API request, both **get\_weather\_by\_coordinates** and **get\_weather\_by\_zip** functions print an error message along with the response status code.

The script also employs logging to record the URLs it fetches data from, the received data, and any errors that occur.

### location.js

This code is responsible for getting the user's current location using the browser's geolocation API and sending it to the server for further processing. Let's break it down:

1. **function getLocation() {...}**: This is a JavaScript function that is triggered when the user clicks the element with the ID **location-btn**.
2. **if (navigator.geolocation) {...}**: This condition checks if the browser supports the geolocation API.
3. **navigator.geolocation.getCurrentPosition(function(position) {...})**: If geolocation is supported, this line retrieves the current position (latitude and longitude) of the user.
4. Inside the callback function, the latitude and longitude values are extracted from the **position** object.
5. An AJAX request is sent to the server using the **XMLHttpRequest** object (**xhr**). The request is sent as a POST request to the **/location** endpoint with the latitude and longitude data included in the request payload as JSON.
6. **xhr.onload** is an event listener that triggers when the AJAX request has completed. If the status code of the response is 200 (indicating a successful request), the page is reloaded using **location.reload()**. This is typically done to update the page with new data received from the server.
7. If the browser doesn't support geolocation, an alert is shown indicating that geolocation is not supported.
8. **window.addEventListener('load', function() {...})**: This event listener waits for the page to finish loading. Once loaded, it attaches an event listener to the element with the ID **location-btn** and calls the **getLocation()** function when that element is clicked.

In summary, this code allows the user to click a button (with ID **location-btn**) to obtain their current location using the browser's geolocation API. The obtained latitude and longitude coordinates are then sent to the server for further processing, potentially retrieving weather information or performing other location-based tasks.

Flask's **url\_for** function is intelligent enough to find the static files. As long as the **location.js** file is in a directory named **static** at the same level as your **templates** directory, Flask will be able to locate the file. The project structure should look something like this:

frizz/

static/

location.js

templates/

frizz\_landing.html

app.py

### frizz\_landing.html

This is an HTML page that represents the user interface for the Frizz Wiz application. It is saved locally in a test environment in **C:\Users\matta\PycharmProjects\frizz\templates** and on the Ubuntu Unix server in **/var/www/frizz/templates/** whereas all of the other files are in the \frizz (/frizz) directory. Let's go through its structure and elements:

The document structure can be broken down into two main parts: the head and the body.

In the **head**:

* It declares the HTML document's language (**lang="en"**), character encoding (**charset="UTF-8"**), and viewport settings for responsive design (**name="viewport"**).
* The **title** tag sets the webpage title as "Frizz Wiz".
* The **style** tag includes CSS styles for various elements on the webpage such as body, form, div, img, and several specific elements with IDs like 'location-btn', 'zip\_code', 'zip\_code\_label', 'submit-btn', 'weather\_photo', and 'resultDiv'. This styling is responsible for the look and layout of the webpage.
* The head also includes scripts for jQuery and jQuery UI, useful for handling events, creating animations, and adding other interactive features.
* It links to a CSS file named **new\_slider.css** from the application's static files.

In the **body**:

* There's a top-level heading (h1) "The Frizz Wiz".
* A form (**id="locationForm"**) which includes two hidden input fields for latitude and longitude, and a button to get the hair forecast.
* A div (**id="resultDiv"**) which is hidden by default, and presumably displayed after a successful fetch of the hair forecast. This div includes a heading, paragraph for displaying results, and an image element for displaying a weather photo.
* Below this, there is a paragraph for extra spacing and a div (**id="slider-container"**), which contains another div (**id="time-range"**). Inside the **time-range** div, a paragraph describes the time range and a slider for users to select the range of time for which they want the forecast.
* At the end, three JavaScript files named **new\_slider.js**, **fetch\_forecast.js**, and **location.js** are linked from the application's static files. These scripts likely contain the JavaScript functionality for the webpage such as fetching the hair forecast, handling slider input, and getting the user's location.

Key elements and attributes such as **{{ color1 }}**, **{{ color2 }}**, **{{ url\_for('static', filename='css/new\_slider.css') }}**, etc. are placeholders which will be filled in with actual data when the HTML is rendered by a web server (like Flask). The placeholder notation **{{ ... }}** is common in web frameworks to signify dynamic content.

The whole webpage is designed for the user to get their hair forecast based on their location and a chosen time range. The forecast results and a corresponding photo will be displayed on the webpage after they have been fetched from the server.

Run\_waitress.py

*note: this file is only needed to run locally, not on a web server*

The code imports the **serve** function from the **waitress** module and the **app** object from the **local\_web\_app** module. Then it uses the **serve** function to start a Waitress server and serve the **app** on a specified host and port. It is only needed when the application is run locally with local\_web\_app (not web\_app which uses gunicorn)

Here's a breakdown of the code:

1. **from waitress import serve**: This line imports the **serve** function from the **waitress** module. Waitress is a production-quality pure-Python server for WSGI applications.
2. **from local\_web\_app import app**: This line imports the **app** object from the **local\_web\_app** module. The **app** object represents your WSGI application. The name **local\_web\_app** is the name of the module containing your application code.
3. **serve(app, host='0.0.0.0', port=8000)**: This line starts the Waitress server and serves your application.
   * **serve**: The **serve** function is called with the **app** object as the first argument.
   * **host='0.0.0.0'**: The **host** parameter specifies the IP address or hostname on which the server should listen. In this case, **'0.0.0.0'** means the server will listen on all available network interfaces.
   * **port=8000**: The **port** parameter specifies the port number on which the server should listen. In this case, it is set to **8000**.

When you execute this code, Waitress will start a server and serve your application at the specified host and port.

### wsgi.py

**wsgi.py**, is an entry point for a WSGI-compatible web server to serve your web application.

WSGI stands for Web Server Gateway Interface. It's a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.

Here's what each part of the script does:

* **from web\_app import app**: This line imports the Flask **app** object from the **web\_app** module. The **app** object is an instance of class Flask. It's the central object of your application and it's used to handle the web requests that your application receives from the web server.
* **if \_\_name\_\_ == "\_\_main\_\_": app.run()**: This is the entry point of your application. If this script is run directly (as opposed to being imported), **\_\_name\_\_** will be **"\_\_main\_\_"**, and **app.run()** will be called. This line starts your application's development server.

It's important to note that **app.run()** is not used when deploying applications to production. When deploying, you'd typically have a WSGI server such as Gunicorn or uWSGI that serves your app.

In the context of a Flask application, a **wsgi.py** file usually sits at the root of the project and is used to expose the Flask application object (**app**) to the WSGI server. The WSGI server uses this object to run the application.

When you use a command like **gunicorn wsgi:app**, it means "run the WSGI application found in the **app** variable in the **wsgi** module".

### fetch\_forecast.js

**fetch\_forecast.js**, is a JavaScript code that fetches weather forecast data from a Flask web application and updates the webpage's content based on the returned forecast. Here's what each part of the script does:

* **window.fetchWeatherForecast = function() {...}**: This line defines a global function **fetchWeatherForecast** which fetches the weather forecast.

Inside the function:

1. **var startHourValue, endHourValue, lat, lon, zipCode**: Variables are declared and initialized to hold the start and end hours of the forecast, latitude, longitude, and Zip code.
2. **var data = {};**: An empty object **data** is created to store the request parameters that will be sent to the Flask app.
3. **var url = '/weather\_general';**: The URL of the endpoint in the Flask app that will provide the forecast data.
4. **fetch(url, {...}).then(function(response) {...}).then(function(json) {...}).catch(function() {...});**: This is the main part of the function where the actual HTTP request is made to the Flask app. It uses the **fetch** API to make a POST request to the Flask app's endpoint, sending the **data** object as the request body.
   * If the request is successful, it updates the text in the **weather\_result** element with the forecast and advice returned by the Flask app.
   * If a photo URL is returned, it updates the source (**src**) of the **weather\_photo** element with the URL and displays the photo. It then makes the **resultDiv** element visible to display the results.
   * If the request fails (either the Flask app returns a non-OK HTTP status, or there's a network error), it updates the text in the **weather\_result** element with an error message.

* **document.addEventListener('load', function() {...});**: This line sets up an event listener to call **fetchWeatherForecast** once the DOM has completely loaded. However, the event name should be **'DOMContentLoaded'** instead of **'load'** to ensure that the DOM is fully loaded before the function is executed. The **load** event waits for all content, including images and other resources, to be completely loaded, which might not be necessary in this case.

### new\_slider.js

**new\_slider.js** sets up a range slider that users can use to select a forecast time period. The slider will display the forecast for the current day from 10 AM to 3 PM by default. When the user adjusts the slider, the timestamps below the slider will update to reflect the new forecast period.

Here's a detailed breakdown of what each part of the script does:

1. **var dt\_from = new Date(); var dt\_to = new Date();**: Declares two variables representing the current date and time (this is the starting and ending time for the slider range).
2. **dt\_to.setHours(dt\_from.getHours() + 24);**: Updates **dt\_to** to be 24 hours from **dt\_from**.
3. **var min\_val = Date.parse(dt\_from) / 1000 / 3600; var max\_val = Date.parse(dt\_to) / 1000 / 3600;**: Converts **dt\_from** and **dt\_to** from Date objects to seconds (since 1970/01/01) then to hours and sets them as the minimum and maximum values for the slider.
4. The **initial\_from** and **initial\_to** variables are set to 10 AM and 3 PM of the current day, respectively.
5. **function zeroPad(num, places) {...}** and **function formatDT(\_\_dt) {...}**: These two functions are helper functions for formatting dates and times. The **zeroPad** function pads a number with zeros on the left to a certain length, and the **formatDT** function formats a Date object into a string in the format of **yyyy-mm-dd hh:mm AM/PM**.
6. **$('.slider-time').html(formatDT(initial\_from)); $('.slider-time2').html(formatDT(initial\_to));**: Updates the HTML content of elements with classes **slider-time** and **slider-time2** with the formatted initial time range.
7. **$("#slider-range").slider({...});**: Initializes a jQuery UI slider with the specified settings.
   * **range: true** means that it's a range slider (i.e., it has two handles).
   * **min: min\_val, max: max\_val** specifies the minimum and maximum values of the slider.
   * **step: 1** means that the slider's values increase or decrease in increments of 1.
   * **values: [initial\_from.getTime() / 1000 / 3600, initial\_to.getTime() / 1000 / 3600]** sets the initial values of the two handles of the slider.
   * The **slide** function updates the HTML content of **.slider-time** and **.slider-time2** with the formatted current range whenever a handle is moved.

Top of Form

### update\_hour\_display.js

**update\_hour\_display.js** defines a function **updateHourDisplay()** that updates the display of start and end hours on the webpage.

Here is what each line does:

1. **var startHourValue = document.getElementById('start\_hour').value; var endHourValue = document.getElementById('end\_hour').value;**: Gets the values of the **start\_hour** and **end\_hour** elements from the HTML document and assigns them to **startHourValue** and **endHourValue** respectively.
2. **var startHourDate = new Date(startHourValue); var endHourDate = new Date(endHourValue);**: Converts the **startHourValue** and **endHourValue** from string to JavaScript Date object.
3. **document.getElementById('start\_hour\_display').innerText = startHourDate.toLocaleString('en-US'); document.getElementById('end\_hour\_display').innerText = endHourDate.toLocaleString('en-US');**: Converts the **startHourDate** and **endHourDate** to string format using **toLocaleString('en-US')** and updates the inner text of the **start\_hour\_display** and **end\_hour\_display** elements.

Finally, an event listener is set up on the **DOMContentLoaded** event, which is fired when the initial HTML document has been completely loaded and parsed. When this event is fired, it calls the **updateHourDisplay** function and also adds event listeners to the **start\_hour** and **end\_hour** elements to call **updateHourDisplay** whenever their values change. This ensures that the displayed time is always up-to-date with the selected start and end hours.

## Firewall & Other Tools

### Firewall (UFW)

The server uses a firewall to limit the rate of incoming connections to the SSH port. UFW is the program

* 1. **If it needs to be reinstalled: sudo apt install ufw**
  2. **It can be enabled with sudo ufw enable**
  3. **By default it blocks all SSH incoming connections and allows all outgoing. To allow incoming connections: sudo ufw allow ssh**
  4. **To check the status of ufw, sudo ufw status verbose**
  5. **To open a port on uft, sudo ufw allow 22**
  6. **To deny connections on uft, sudo ufw deny 22**
  7. **This will prevent all connections including Putty**

Allowing SSH access to your server from the internet can be a security risk. Ensure that you're following best practices for securing SSH, such as using key-based authentication, disallowing root login, and potentially using a non-standard port.

Key-based authentication has been used. My identification has been saved in C:\Users\matta/.ssh/id\_rsa.

Your public key has been saved in C:\Users\matta/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:QC2Bd9HyH4fmp4F5PvNxEhNNkE6Eqid/VousUuGJFLc alevy.matthew@gmail.com

The key's randomart image is:

+---[RSA 4096]----+

| .oo.o ooo.|

| ..o = o . oo |

| ..o = o +. .|

| .. E + o. |

| .S+ O oo |

| + B = oo |

| = + \*o..|

| . . X .+ |

| ..+ +. |

+----[SHA256]-----+

1. **Fail2Ban:** Fail2Ban is a tool that can automatically update firewall rules to ban IP addresses that make too many failed login attempts.
   1. If it needs to be reinstalled: **sudo apt install fail2ban**
   2. To start and enable it: **sudo systemctl start fail2ban and sudo systemctl enable fail2ban**
2. Because of attempts to guess the login password (server might be under a brute-force attack, where an attacker is trying to guess your root password), the root login has been disabled.
   1. To change this, edit the SSH config file: sudo nano **/etc/ssh/sshd\_config**
   2. Find the line that says PermitRootLogin and it was changed to no. If the line doesn't exist, you can add it to the file. Then save and exit the file.
   3. Then, restart SSHD: **sudo systemctl restart ssh**

We created a new user with sudo privileges before disabling root login, and tested logging in as this user in a new terminal session before closing the current one. The new user is matt0164.

These steps should help increase the security of your server and might alleviate the load that these failed login attempts are putting on your server, allowing you to restart Nginx and check its status.

### Virtual Environment

A virtual environment is an isolated environment where you can install Python packages without interfering with packages installed in other environments or the system's global Python environment. This can be especially useful when different projects require different versions of the same package.

To start using a virtual environment, you have to activate it. You can do that using the following command:

1. Activate the virtual environment: **source /var/www/frizz/venv/bin/activate**

Here is how to create a new virtual environment:

1. Navigate to your project directory (e.g., **/var/www/frizz**).
2. Create a new virtual environment: **python3 -m venv venv**
3. This will create a new directory named **venv** in the current directory.

After doing this, the correct path for the **Environment** directive should be **/var/www/frizz/venv/bin**.

Also, please note that in the **ExecStart** directive, the Gunicorn executable is being referenced from **/usr/local/bin/gunicorn**. However, when using a virtual environment, Gunicorn should be installed in the virtual environment and the path should be the one inside the virtual environment. So it should be something like **/var/www/frizz/venv/bin/gunicorn**.

If you need to re-install gunicorn, it should be done within the virtual environment using the following command:

1. Create a new virtual environment

**python3 -m venv /path/to/new/virtual/environment**

1. Install Gunicorn:

**pip install gunicorn**

## The Algorithm

1. **Determining the Forecast Period:**
   * By default, the program will provide a weather forecast for the 'upcoming day' based on the time the user checks the app.
     + If checked in the morning (let's say before 10 AM), the app will provide a forecast from 8 AM that day to 8 PM of the same day.
     + If checked in the evening (let's say after 10 AM), the forecast will span the remainder of that day and the entirety of the next day.
   * The program will also provide a 7-day forecast, allowing users to select a different day or time period.
2. **Highlighting Critical Periods:**
   * The app will analyze the forecast data for the next 24 and 72 hours to identify any 'critical periods' - times when humidity or wind speed will be extremely high or low.
   * Users will be notified about these critical periods, which can help them plan their hair care and styling accordingly.
3. **Weekly Snapshot:**
   * In addition to the daily forecast, the app will provide a snapshot for the next 5 to 7 days.
   * This snapshot will highlight any particularly good or bad days for certain hairstyles based on the expected humidity and wind speed.
   * For example, it could tell a user "next Tuesday will be a great day for straight hair" or "next Thursday will be a very tough day for straight hair due to high humidity and wind speeds".
4. **User-Centric Customizations:**
   * Users can adjust their 'hair check' times according to their routine, enabling the app to provide a more personalized forecast.
   * Based on the forecast and the user's specified schedule, the app will offer custom recommendations on the optimal hair care/styling for that day.

## Quick tasks

1. To switch to the main directory

**cd /var/www/frizz**

1. To activate the virtual environment

**source /var/www/frizz/venv/bin/activate**

1. To edit the service file that starts gunicorn

**sudo nano /etc/systemd/system/frizzapp.service**

1. To restart the gunicorn frizz.service

**sudo systemctl restart frizzapp.service**

1. To check the status of the service to ensure it's running in bash:

**sudo systemctl status frizzapp.service**

1. To review the status of the gunicorn service

**sudo journalctl -u frizzapp.service --since "2 minutes ago"**

1. To edit nginx config

**sudo nano /etc/nginx/sites-available/default**

1. To restart ngnix and test its status

**sudo systemctl restart nginx**

**sudo nginx -t**

1. To clone the latest branch from Github

**cd /var/www/frizz**

**git clone git@github.com:matt0164/frizz.git**

**git fetch origin**

**git reset --hard origin/latest**

**DO NOT USE: git clone -b latest** [**git@github.com:matt0164/frizz.git**](mailto:git@github.com:matt0164/frizz.git)

1. To check that files were updated (replace filename)

**ls -l --time=status web\_app.py**

# Configuration & Troubleshooting

## Gunicorn

Gunicorn is installed on the server (using **sudo pip3 install gunicorn**)

in the following path /usr/local/bin/gunicorn (to check, run **which gunicorn**)

To test Gunicorn :

1. Activate the virtual environment:

**source /var/www/frizz/venv/bin/activate**

1. Navigate to the project directory:

**cd /var/www/frizz/**

1. Test gunicorn

**gunicorn wsgi:app**

*note: leave off the .py when running web\_app using gunicorn*

Gunicorn should run and show some output like this:

[2023-05-17 08:27:27 +0000] [42262] [INFO] Starting gunicorn 20.1.0

[2023-05-17 08:27:27 +0000] [42262] [INFO] Listening at: http://127.0.0.1:8000 (42262)

[2023-05-17 08:27:27 +0000] [42262] [INFO] Using worker: sync

[2023-05-17 08:27:27 +0000] [42263] [INFO] Booting worker with pid: 42263

If there are errors, it can often be the following:

1. Permission errors to ensure that Gunicorn can log errors. If you get this, change the ownership of the directory and the file to:

sudo chown frizzwizceo:frizzwizceo /var/www/frizz/logs -R

This command will recursively (**-R**) change the ownership of the directory **/var/www/frizz/logs** and its contents to the user **frizzwizceo** and the group **frizzwizceo**. Now try starting your Gunicorn process again.

1. When you run the system process, sometimes the system can't find Gunicorn at the specified location **/var/www/frizz/venv/bin/gunicorn**

Here are a few troubleshooting steps:

1. Ensure that the virtual environment has been activated before running Gunicorn. If it hasn't been, then Gunicorn might not be available in the shell's PATH.
2. Verify that Gunicorn is installed in the virtual environment. You can do this by activating the virtual environment and running **which gunicorn**:

This command will print the path to the Gunicorn executable if it's installed.

If Gunicorn is not installed, you can install it using pip within your virtual environment:

source /var/www/frizz/venv/bin/activate pip install gunicorn

1. If Gunicorn is installed but the path is different from what's in the **frizzapp.service** file, then you'll need to update the service file with the correct path.

You can edit the **frizzapp.service** file using a text editor like nano:

sudo nano /etc/systemd/system/frizzapp.service

In the file, change the path in the line that starts with **ExecStart** to match the path returned by the **which gunicorn** command, then save and exit the file. After this, reload the systemd daemon and restart your service:

sudo systemctl daemon-reload sudo systemctl restart frizzapp.service

1. Make sure the permissions on the **venv** directory allow the **frizzwizceo** user to read and execute files. You can ensure this with the following command:

sudo chmod -R 755 /var/www/frizz/venv

This command will recursively (**-R**) change the permissions of the directory **/var/www/frizz/venv** and its contents to allow read, write, and execute permissions for the owner, and read and execute permissions for the group and others.

1. The error **Connection in use: ('127.0.0.1', 8000)** indicates that the port 8000 is currently being used by another process.
   1. You can use the following command to find out what process is using port 8000:

sudo lsof -i :8000

* 1. This command will show you a list of all processes using the port. The output will look something like this:

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

gunicorn 138412 frizzwizceo 5u IPv4 1515041 0t0 TCP localhost:8000 (LISTEN)

gunicorn 138413 frizzwizceo 5u IPv4 1515041 0t0 TCP localhost:8000 (LISTEN)

* 1. In this output, the second column (PID) shows the process ID of the process that's using the port. You can stop this process using the kill command and the PID:

sudo kill -9 <PID>

Replace **<PID>** with the process ID from the output of the **lsof** command.

After killing the process, try restarting the **frizzapp.service** again.

You can paste any errors into Chat GPT to troubleshoot further.

1. you can stop it with **Ctrl+C** and proceed to the next step.
2. **Create a Gunicorn systemd Service File:** We have created a systemd service file to start and manage the Gunicorn process. First, we created the file:

To edit the Gunicorn system file,

SSH:

**sudo nano /etc/systemd/system/frizzapp.service**

This file should contain the following:

**[Unit]**

Description=Gunicorn instance to serve frizzapp

After=network.target

**[Service]**

User=frizzwizceo

Group=www-data

WorkingDirectory=/var/www/frizz/

Environment="PATH=/var/www/frizz/venv/bin/"

ExecStart=/usr/local/bin/gunicorn --workers 3 --bind 127.0.0.1:8000 wsgi:app --access-logfile '-' --error-logfile '-'

**[Install]**

WantedBy=multi-user.target

If you make any changes, run:

**sudo systemctl daemon-reload**

**sudo systemctl restart frizzapp.service**

Then check the status of the service:

### sudo systemctl status frizzapp.service

### To stop any running Gunicorn and Nginx processes, you can use the following commands. Note that you'll need to have superuser privileges to run these commands, which usually means you'll need to include sudo at the beginning of each command.

### sudo systemctl stop frizz.service

### sudo systemctl stop nginx

### Once you've stopped these services, you can check the status of the ports to see if they're ready to listen. You can use the ss command to do this:

### sudo ss -tuln

The command options **-tuln** are for:

* **t**: TCP ports
* **u**: UDP ports
* **l**: only services which are listening on some port
* **n**: show port number, don't try to resolve the service name

After you run this command, you should see a list of ports and the services that are using them. Look for ports 80 and 8000, they should be free now that we have stopped Nginx and Gunicorn.

### Check the service file by opening it in nano:

### sudo nano /etc/systemd/system/frizzapp.service

### A backup of this file will contain the correct configuration (see page 18)

### Then activate the virtual environement

### If you need to make any changes, reload the system daemon to apply the changes and restart and check the service.

## Nginx

### Basic Guide to Nginx Configuration

1. **Understand Nginx Configuration File Structure**: Nginx configuration files are typically stored in /etc/nginx. Here are the key files and directories:
   * **nginx.conf**: This is the main Nginx configuration file.
   * **sites-available**: This directory contains configuration files for each server block (i.e., website) you want to host.
   * **sites-enabled**: This directory contains links to the configuration files for the server blocks (websites) you want to enable.
   * **conf.d**: This directory is used to store additional configuration files.
   * **mime.types**: This file maps file extensions to their MIME type.
2. **Edit Configuration Files**: The configuration files can be edited using any text editor such as nano, vim, etc. Always use 'sudo' to ensure you have the necessary permissions.

**sudo nano /etc/nginx/sites-available/default**

1. **Check Configuration for Errors**: Always check your configuration for any errors before you try to restart or reload Nginx. You can do this using the **-t** flag:

**sudo nginx -t**

If there are no errors, you should see the message "configuration file /etc/nginx/nginx.conf test is successful".

1. **Apply Configuration Changes**: After editing the configuration and checking for errors, you need to reload or restart Nginx to apply the changes. This can be done using the systemctl command:

**sudo systemctl reload nginx**

or

**sudo systemctl restart nginx**

### Troubleshooting Common Nginx Configuration Issues

1. **403 Forbidden Error**: This usually indicates a permissions problem. Check that Nginx has read access to the files and directories in your website's root directory. Also, ensure that the directory itself is executable.
2. **502 Bad Gateway Error**: This error can occur if Nginx is unable to communicate with your Flask app or any other upstream server it's proxying requests to. Check that your upstream servers are running and that the proxy settings in your Nginx configuration are correct.
3. **"Failed to read PID from file" Error**: This error can occur if Nginx is unable to write to the PID file specified in your nginx.conf file. Check that the directory where the PID file is located exists and that Nginx has write access to it.
4. **Changes Not Applied**: If you made changes to your configuration and they don't seem to be applied, check that you edited the correct configuration file and that you reloaded or restarted Nginx afterwards. Also, use **sudo nginx -t** to check for syntax errors in your configuration files.
5. **Nginx Doesn't Start After Reboot**: This can happen if Nginx isn't enabled to start at boot. You can fix this by running **sudo systemctl enable nginx**.
6. **Check Nginx/Aplication Logs**: When encountering issues, checking Nginx logs (**/var/log/nginx/error.log** or **/var/log/nginx/access.log**) and your application logs could provide useful insight into the cause of the issue.

Remember to keep backups of your original configuration files before making any changes, as it can help revert any undesired changes. Always test new configuration files before applying them.

### Frizz App Nginx Configuration

The Nginx configuration for the application is stored in the file **/etc/nginx/sites-available/default** and **/etc/nginx/nginx.conf**. The file has include statements that also reference **/etc/nginx/mime.types; /etc/nginx/conf.d/\*.conf; and /etc/nginx/sites-enabled/\*;** These configuration files specify how incoming HTTP requests are handled by the server.

The server listens on port 80, and it is configured to proxy requests to the application running on **localhost** (127.0.0.1) port 8000. It also includes headers to provide information about the original client request, as it proxies the request to the application.

The server is also configured to serve the directory **/.well-known/acme-challenge/** from **/var/www/html**, which is used for the Let's Encrypt SSL certificate verification process.

The nginx config file should contain the following:

server {

listen 80;

server\_name app.frizz-wiz.com;

location / {

return 301 https://$host$request\_uri;

}

}

server {

listen 443 ssl;

server\_name app.frizz-wiz.com;

ssl\_certificate /etc/letsencrypt/live/app.frizz-wiz.com/fullchain.pem;

ssl\_certificate\_key /etc/letsencrypt/live/app.frizz-wiz.com/privkey.pem;

location / {

include proxy\_params;

proxy\_pass http://unix:/tmp/frizz/frizz.sock;

}

location = /favicon.ico { access\_log off; log\_not\_found off; }

location /static/ {

root /var/www/frizz;

}

}

**Changing the Configuration**

If you need to change the server configuration, edit the file **/etc/nginx/sites-available/default** with a text editor (like **nano** or **vim**). See the section called **nginx config** for the current/correct config. You'll need **sudo** privileges to edit this file. After making changes, save the file and exit the text editor.

**sudo nano /etc/nginx/sites-available/default**

Next, test the new configuration by running **sudo nginx -t**. If this command reports that the configuration is okay, reload the Nginx configuration by running **sudo systemctl reload nginx**.

To view the error log in nginx, you can

1. Open SSH log in and enter this command:

**sudo nano /var/log/nginx/error.log**

## SSL Certificate

The application uses an SSL certificate to secure connections. This certificate is provided by Let's Encrypt, a free, automated, and open certificate authority. The certificate needs to be renewed every 90 days.

If you encounter any issues while renewing the certificate or if the SSL certificate was purchased from a different provider (like GoDaddy), follow the certificate provider's instructions for renewal and installation.

Location of key files:

frizz\_landing.html /var/www/frizz/web\_app/templates

The code can be updated via Git using a public/private key. The key is saved in /home/frizzwizceo/.ssh/id\_rsa as first-key.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDC1UTLqblrHtgdLm4tNACjaOIg5dHR90ek3OleYn3>

## Encapsulated Summary (for ChatGPT)

Frizz-Wiz is a web application running on an Nginx server. The application is configured to be accessible from the internet via the domain names frizz-wiz.com and www.frizz-wiz.com.

Nginx Configuration is stored in the file /etc/nginx/sites-available/default and /etc/nginx/nginx.conf. The file has include statements that also reference /etc/nginx/mime.types; /etc/nginx/conf.d/\*.conf; and /etc/nginx/sites-enabled/\*; These configuration files specify how incoming HTTP requests are handled by the server. The server listens on port 80, and it is configured to proxy requests to the application running on localhost (127.0.0.1) port 8000. It also includes headers to provide information about the original client request, as it proxies the request to the application. The server is also configured to serve the directory /.well-known/acme-challenge/ from /var/www/html, which is used for the Let's Encrypt SSL certificate verification process.

**My flask web\_app**

Various Python scripts are used in the Frizz Wiz application to handle different tasks. These scripts include functions for retrieving weather data, geolocation information, and generating random colors. They interact with external APIs, such as OpenWeatherMap and IPinfo, to fetch weather data and geolocation details. The scripts are called by the Flask application to provide the necessary data for rendering the web pages.

The HTML templates define the structure and content of the web pages displayed to users. They utilize template engines, such as Jinja2, to dynamically insert data retrieved from the Python scripts. The templates include forms for user input, display weather results, and render appropriate images based on weather conditions. JavaScript is also used to handle geolocation functionality and make AJAX requests to the server.

The Flask application is the core of the Frizz Wiz web server. The application interacts with the Gunicorn server to serve the web pages to clients. Gunicorn is responsible for running the Flask application as a web server. By combining these components, the Frizz Wiz application functions as a web server that receives user input, retrieves weather data, and dynamically renders HTML pages with the corresponding information.

The project is saved locally at C:/Users/matta/PycharmProjects/frizz/

The project is served from my GoDaddy VPS server with an IP address is **186.184.30.145**

I am running Ubuntu 22.04 and the project is served from **/var/www/frizz**

Nginx is installed in **/usr/sbin/nginx /usr/lib/nginx /etc/nginx /usr/share/nginx /usr/share/man/man8/nginx.8.gz**

Gunicorn is installed in **/var/www/frizz/**

My systemmd service file for gunicorn should be: **/etc/systemd/system/frizz.service**

My nginx error log is in **/var/log/nginx/error.log**

The application uses an SSL certificate to secure connections. This certificate is provided by Let's Encrypt

Location of key files:

frizz\_landing.html /var/www/frizz/web\_app/templates

My code can be updated via Git using a public/private key. The key is saved in /home/frizzwizceo/.ssh/id\_rsa as first-key.pub and the key is:

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDC1UTLqblrHtgdLm4tNACjaOIg5dHR90ek3OleYn3>

1. web\_app.py: This Python file serves as the entry point for the Flask application. It imports the necessary modules, sets up the Flask app, defines the routes, and handles the logic for retrieving weather data based on zip code or location. The routes interact with the HTML templates by rendering them and passing data from the Python scripts to be displayed dynamically. Additionally, it interacts with the JavaScript file location.js by receiving geolocation data via AJAX requests and triggering a page reload after processing.

2. web\_main.py: This Python file contains functions for fetching weather data from external APIs based on zip codes, location coordinates, or city names. The get\_weather\_by\_zip() function retrieves weather data using the OpenWeatherMap API and performs calculations to determine if it's a good or bad hair day. The get\_weather\_by\_lat\_lng() function does the same but uses latitude and longitude coordinates. These functions return the weather result and photo URL to be displayed in the HTML templates.

3. wsgi.py: This Python file is the entry point for the application when it's run with a WSGI server like Gunicorn. It imports the Flask application instance (app) from the web\_app module. This app is expected to be a Flask instance. It checks if this script is being run directly or being imported. The code inside this block runs only when the script is run directly. It starts the Flask's built-in development server.

4. location.js: This JavaScript file is responsible for handling geolocation functionality in the web page. It checks if the browser supports geolocation and, if so, retrieves the current latitude and longitude coordinates. It then sends this data to the server using an AJAX request to the /location route. Upon receiving a successful response (status code 200), it triggers a page reload to display the updated weather information. This file connects with the Python script web\_app.py through the AJAX request and response handling.

5. frizz\_landing.html: This HTML template is the main page of the Frizz Wiz application. It includes a form where users can enter a zip code or click a button to use geolocation. The form submits a POST request to the / route in the Flask application. It also includes placeholders for displaying the weather result, photo, and dynamically generated colors. The template interacts with the Python script web\_app.py by receiving the weather result and photo data to be rendered in the appropriate sections.