# [Headers / Python]

## General Information & Licensing： Flask

| Code Repository | [pallets/flask](https://github.com/pallets/flask.git) |
| --- | --- |
| License Type | three clause BSD License |
| License Description | * Allow to modify * Allow to distribute * Allow to place warranty |
| License Restrictions | * Not allow to use trademark * Not allow to hold liable |

## General Information & Licensing： Flask-SocketIO

| Code Repository | [GitHub - miguelgrinberg/Flask-SocketIO: Socket.IO integration for Flask applications.](https://github.com/miguelgrinberg/Flask-SocketIO) |
| --- | --- |
| License Type | MIT License |
| License Description | * Allow to modify * Allow for Commercial use * Allow to distribute * Allow for Private use |
| License Restrictions | * Not allow to use trademark * Not allow to hold liable |

## General Information & Licensing： werkzeug

| Code Repository | [werkzeug/LICENSE.rst at main](https://github.com/pallets/werkzeug/blob/main/LICENSE.rst) |
| --- | --- |
| License Type | MIT License |
| License Description | * Allow to modify * Allow for Commercial use * Allow to distribute * Allow for Private use |
| License Restrictions | * Not allow to use trademark * Not allow to hold liable |

## 

| Flask will provide users with the tools needed for building  a web application or a response cycle. The application will receive user requests and send back a response based on the function in the form of HTML, JSON, XML, etc.  https://github.com/pallets/flask/blob/main/src/flask/app.py  Flask-SocketIO gives Flask applications access to low latency bi-directional communications between the clients and the server. The tools from this library provide a web frame for us to establish a permanent connection to the server.  We start the web server with socketio’s run method  [Flask-SocketIO/\_\_init\_\_.py at main](https://github.com/miguelgrinberg/Flask-SocketIO/blob/main/src/flask_socketio/__init__.py#L551)  Under socketio class, it reads the options from the route given and fulfills the required elements for \_\_init\_\_  [Flask-SocketIO/\_\_init\_\_.py at main](https://github.com/miguelgrinberg/Flask-SocketIO/blob/main/src/flask_socketio/__init__.py#L171)  The run method will read the async\_mode and choose how to set the connection.  In our case, our async\_mode is “threading”, it calls the **flask run** method after a few implementation checks.  **FLASK.run:** [flask/app.py at main · pallets/flask · GitHub](https://github.com/pallets/flask/blob/main/src/flask/app.py#L1067)  This calls **run\_simple** with the route host, port, and option.  **run\_simple**: [flask/app.py at main · pallets/flask · GitHub](https://github.com/pallets/flask/blob/main/src/flask/app.py#L1191)  Under run\_simple, it imprement an parameter name “request\_handler” which is a **WSGIRequestHandler** class.  **WSGIRequestHandler:**[werkzeug/serving.py at main](https://github.com/pallets/werkzeug/blob/main/src/werkzeug/serving.py#L148)  **WSGIRequestHandler** provides a basic HTTP/1.1 Web server for WSGI applications. This required **BaseHTTPRequestHandler** class to function. The required parameters are inputted when we created the server(check TCP reports for details).  **BaseHTTPRequestHandler:** [cpython/server.py at main · python/cpython · GitHub](https://github.com/python/cpython/blob/main/Lib/http/server.py#L146)  This is for implementing HTTP servers and handling the HTTP and header parsing. |
| --- |
| When a request is received, we call the **handle()** method and jump to the **handle\_one\_request()** method in both **BaseHTTPRequestHandler** classes. This is used to handle HTTP requests and what it does is it first reads the request and checks the connection, if everything passes, it starts parsing the header by calling **parse\_request**.  **handle\_one\_request:** [cpython/server.py at main · python/cpython · GitHub](https://github.com/python/cpython/blob/main/Lib/http/server.py#L391)  **parse\_request**：[cpython/server.py at main · python/cpython · GitHub](https://github.com/python/cpython/blob/main/Lib/http/server.py#L267)  The method uses rstrip and split to process the splitting on the **\r\n**. This gets the protocol version and filters the unprocessable HTTPS status. It then parses the header by calling **http.client.parse\_headers**  **parse\_headers：**[**cpython/client.py at main**](https://github.com/python/cpython/blob/main/Lib/http/client.py#L224)  The **parse\_headers** function has three parts  1. **\_read\_headers**：[cpython/client.py at main](https://github.com/python/cpython/blob/main/Lib/http/client.py#L206)  reads and extracts the header lines by looping and adding to an empty list from the request line until it hits  one of **[b'\r\n', b'\n', b' ']**  2. Set up the string header for parsing  3. **parsestr：**[**cpython/parser.py at main - email**](https://github.com/python/cpython/blob/main/Lib/email/parser.py#L41)  Create a message structure from a string and returns the root of the message structure.  After the parsing is done, it goes back to **handle\_one\_request** and handles the rest using the parsed header. |