# [TCP / 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 in this case a response cycle. The application will receive requests  from the users and sends back a response based on the function in the form of HTML, JSON, XML, etc.  [flask/app.py at main · pallets/flask · GitHub](https://github.com/pallets/flask/blob/main/src/flask/app.py)  Flask is imported into the /app.py directory. What we currently have is multiple python  decorator which checks and gives a response based on the route and the request type.  The decorator we use to check URLs is route()  route(): it is a decorator to tell Flask what URL should trigger our function.  It also calls add\_url\_rule() which sets the view function to the endpoint.  route()  [flask/scaffold.py at main · pallets/flask · GitHub](https://github.com/pallets/flask/blob/main/src/flask/scaffold.py#L423)  add\_url\_rule()  [flask/scaffold.py at main · pallets/flask · GitHub](https://github.com/pallets/flask/blob/main/src/flask/scaffold.py#L455)  This will handle and set up the routes, for example,  It checks if the method is GET or not, it also adds OPTION  Check None for view\_func and if false, check if the endpoint associate.  Check if view\_func has required\_methods, added to the method if True |
| --- |
| 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)  This method takes the hostname and port number to bind to, as well as the WSGI application and options such as the debug flag and SSL context. During the process, it calls **make\_server** from werkzeug library with its input parameter  **make\_server:** [werkzeug/serving.py at main](https://github.com/pallets/werkzeug/blob/main/src/werkzeug/serving.py#L853)  uses the data carried from **run\_simple()** to settle an **HTTPServer** class (python3 library) and create a **BaseWSGIServer.**  **HTTPServer:** [cpython/server.py at main · python/cpython · GitHub](https://github.com/python/cpython/blob/main/Lib/http/server.py#L130)  This concludes **TCPServer** class as a parameter which itself requires a **BaseServer** class  **BaseWSGIServer:**[werkzeug/serving.py at main](https://github.com/pallets/werkzeug/blob/main/src/werkzeug/serving.py#L651)  The server will handle the request including setting the handler and HTTP protocol.  This also includes **serve\_forver** which loops and continuously checks for new and processes incoming TCP requests until an explicit shutdown() request.  **serve\_forver:** [werkzeug/serving.py at main](https://github.com/pallets/werkzeug/blob/main/src/werkzeug/serving.py#L764)  This **serve\_forver** jump form werkzeug to **serve\_forver()** in the socketserver folder under python library.  **serve\_forever():**[**cpython/socketserver.py at main**](https://github.com/python/cpython/blob/main/Lib/socketserver.py#L216)  This first calls **selector.select()** and wait for events on a set of the registered file object. Once it is ready, the function calls **\_handle\_request\_noblock()** to handle single HTTP request without blocking the server.  **\_handle\_request\_noblock():**[**cpython/socketserver.py at main**](https://github.com/python/cpython/blob/main/Lib/socketserver.py#L303)  If **verify\_request** verifies that an HTTP request is valid, it will be passed to **process\_request. process\_request** jumps to **finish\_request** and creates a **RequestHandlerClass** which is under \_\_init\_\_ of **BaseServer.** We can use this class since we had **TCPServer** set up when hitting  **HTTPServer**  **RequestHandlerClass** will work as the subclass of the **BaseHTTPRequestHandler** class so the HTTPServer can call its **handle()** which handles multiple requests if necessary. |