Open-Source Report

[Flask-Sock]

General Information & Licensing

Code Repository	https://github.com/Ai-Jesse/Blue-Jesse
License Type	MIT
License Description	• Any user can obtain this software free of charge • Any user is allowed to copy, modify, and distribute • Users are allowed to sell the software
License Restrictions	• The software is provided “as is” without warrant • The authors or copyright holders can’t be liable for any claim, damages, or other liability

Magic 

*This section will likely grow beyond the page

In this project, we decided to use Flask-Sock instead of Flask-SocketIO, which provides modern WebSocket support for our application. And this extension different than others is that it works with the Flask development web server, without the need to install a greenlet based server suck as gevent or eventlet. More flexible.

```
29    app = Flask(name)
30    print("App running I think")
31    sock = Sock(app)
```

app = Flask(name), from the flask packages. It creates an instance of Flask. **(name**) is the name of the current Python module. The application needs to know where it is in order to set some paths, and name is a convenient way to tell it.
https://github.com/pallets/flask/blob/main/src/flask/app.py#L553

sock = Sock(app), from the flask_sock packages. https://github.com/miguelgrinberg/flask-sock/blob/10e0b3bb05580b63433106ef1b5ab42a735846d6/src/flask_sock/__init__.py#L6
The extension is added to our Flask application by creating a **Sock()** class instance. Because we have a global app object(**app = Flask(name)**), we can use the direct initialization method.

```
@sock.route("/singleplayer")
def ws_singleplayer(ws):

    #create a singleplayer game, will keep recieving data and pass it to the game object, handle the code in game class
    game = SingleGame(ws)
    while True:
        data = ws.receive()
        game.handle(data)
```

The sock instance has a route decorator, that works pretty much like Flask's own, but it adds the WebSocket protocol handshake so that the route can speak WebSocket instead of HTTP.

In our **game.py,**

```
4     class SingleGame():
          Just-Google
5         def __init__(self, socket):
6             self.socket = socket
7
          Just-Google
8         def handle(self, data):
9             self.socket.send(data)
10
```

the **handle()** we make will replace the common **ws.send()** function from all WebSocket parts.

–

_init__

https://github.com/miguelgrinberg/flask-sock/blob/10e0b3bb05580b63433106ef1b5ab42a735846d6/src/flask_sock/__init__.py#L12

```
12      def __init__(self, app=None):
13          self.app = None
14          self.bp = None
15          if app is None:
16              self.bp = Blueprint('__flask_sock', __name__)
17          else:
18              self.app = app
19              self.init_app(app)
20
```

Inside the Sock class, the **__init__** method. This method's parameters are the class instance (self) and the Flask instance. It sets both the **self.app and bp(Blueprint)**' instances as None. A blueprint defines a collection of views, templates, static files and other elements that can be applied to an application. If the Flask instance is empty, the bp class will be defined as **Blueprint('__flask_sock', __name__)**. If Flash instance is not empty, it return to the instance that it has as **self.app = app.**

For the WebSocket server,

```
208 @sock.route("/singleplayer")
209 def ws_singleplayer(ws):
210
211     #create a singleplayer game, will keep recieving data and pass it to the game object, handle the code in game class
212     game = SingleGame(ws)
213     while True:
214         data = ws.receive()
215         game.handle(data)
216
```

The **ws** object passed to the route is the actual WebSocket connection, and the function can exchange information with the client via the **ws.receive()** and **handle(send) ()** methods. It goes without saying that when multiple clients connect at the same time, each client gets its own **ws** object, and the communication between the server and each client is private.

The **sock.route** decorator is fully integrated with Flask.

https://github.com/miguelgrinberg/flask-sock/blob/10e0b3bb05580b63433106ef1b5ab42a735846d6/src/flask_sock/__init__.py#L32

```
53          def decorator(f):
54              @wraps(f)
55              def websocket_route(*args, **kwargs):  # pragma: no cover
56                  ws = Server(request.environ, **current_app.config.get(
57                      'SOCK_SERVER_OPTIONS', {}))
58                  try:
59                      f(ws, *args, **kwargs)
60                  except ConnectionClosed:
61                      pass
62                  try:
63                      ws.close()
64                  except:  # noqa: E722
65                      pass
66
67                  class WebSocketResponse(Response):
68                      def __call__(self, *args, **kwargs):
```

It does define the server that WebSocket use. Which be defined in **class Server(Base):**

https://github.com/miguelgrinberg/simple-websocket/blob/main/src/simple_websocket/ws.py#L253

This class implements a WebSocket server. More details can be seen in **__init__**,

https://github.com/miguelgrinberg/simple-websocket/blob/main/src/simple_websocket/ws.py#L288

```
288     def __init__(self, environ, subprotocols=None, receive_bytes=4096,
289                  ping_interval=None, max_message_size=None, thread_class=None,
290                  event_class=None, selector_class=None):
291         self.environ = environ
292         self.subprotocols = subprotocols or []
293         if isinstance(self.subprotocols, str):
294             self.subprotocols = [self.subprotocols]
295         self.mode = 'unknown'
296         sock = None
```

For those different parameters, **(environ，subprotocols，receive_bytes，ping_interval，max_message_size，thread_class，event_class，selector_class).**

environ, A WSGI ``environ`` dictionary with the request details. Among other things, this class expects to find the low-level network socket for the connection somewhere in this dictionary**. Werkzeug, Gunicorn, Eventlet and Gevent are the only web servers that are currently supported.** Can also see that inside the **__init__. Set sock to empty, search four WSGI to environ** in **class Server(Base),** the time environ get which **WSGI** is using, will change the model to fit.

After above, WebSocket class is established.

decorator methond

https://github.com/miguelgrinberg/flask-sock/blob/10e0b3bb05580b63433106ef1b5ab42a735846d6/src/flask_sock/__init__.py#L53

```
53          def decorator(f):
54              @wraps(f)
55              def websocket_route(*args, **kwargs):  # pragma: no cover
56                  ws = Server(request.environ, **current_app.config.get(
57                      'SOCK_SERVER_OPTIONS', {}))
58                  try:
59                      f(ws, *args, **kwargs)
60                  except ConnectionClosed:
61                      pass
62                  try:
63                      ws.close()
64                  except:  # noqa: E722
65                      pass
66
67                  class WebSocketResponse(Response):
68                      def __call__(self, *args, **kwargs):
69                          if ws.mode == 'eventlet':
70                              try:
71                                  from eventlet.wsgi import WSGI_LOCAL
72                                  ALREADY_HANDLED = []
73                              except ImportError:
74                                  from eventlet.wsgi import ALREADY_HANDLED
75                                  WSGI_LOCAL = None
76
77                              if hasattr(WSGI_LOCAL, 'already_handled'):
78                                  WSGI_LOCAL.already_handled = True
79                              return ALREADY_HANDLED
80                          elif ws.mode == 'gunicorn':
81                              raise StopIteration()
```

Inside the decorator method, the **websocket_route** method leads to the base server for the websocket with: **ws = Server(request.environ, **current_app.config.get('SOCK_SERVER_OPTIONS', {})).**

The **WebSocketResponse** class is the important class which let websocket server receive the response. https://github.com/miguelgrinberg/flask-sock/blob/10e0b3bb05580b63433106ef1b5ab42a735846d6/src/flask_sock/__init__.py#L67

For the call function,

```
def __call__(self, *args, **kwargs):
    if ws.mode == 'eventlet':
        try:
            from eventlet.wsgi import WSGI_LOCAL
            ALREADY_HANDLED = []
```

if the WebSocket mode is “eventlet”, it goes to a empty list, it no **ImportError** shows, the ws.mode return to eventlet.

In our snake game, server will be able to get connection by using ws.send() and ws.receive() with users(client). From all moving and getting score by single or multi players, the websocket server will receive the data to update the leaderboard.