Open-Source Report

[Flask-Sock]

General Information & Licensing

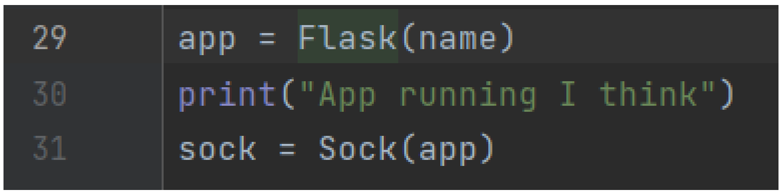
Code Repository	https://github.com/Ai-Jesse/Blue-Jesse
License Type	MIT
License Description	• Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: • The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. • THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION

	WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
License Restrictions	• The original copyright notice. • A copy of the license itself

Magic ★★。˚⋆｡˚☽˚｡⋆★⋆｡˚★彡★ ༄

*This section will likely grow beyond the page

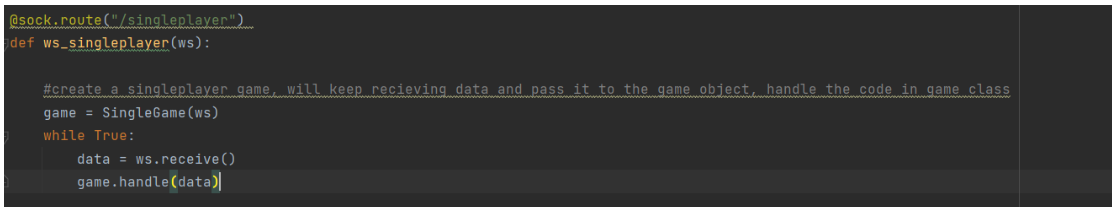
In this project, we decided to use Flask-Sock instead of Flask-SocketIO, which provides modern WebSocket support for our application. And this extension different than others is that it works with the Flask development web server, without the need to install a greenlet based server suck as gevent or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



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

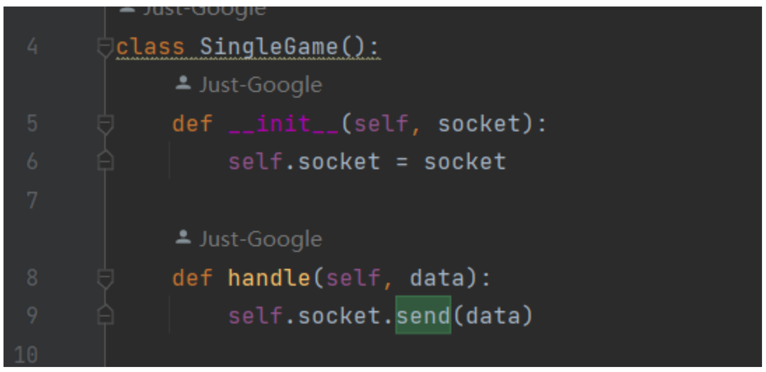
app = Flask(name), from the flask packages. It creates an instance of Flask. **(name**) is the name of the current Python module. The application needs to know where it is in order to set some paths, and name is a convenient way to tell it.
https://github.com/pallets/flask/blob/main/src/flask/app.py#L553

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,** 

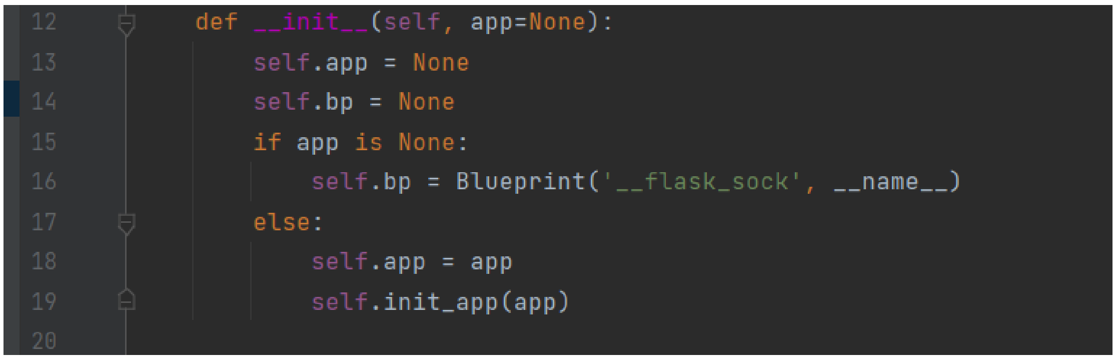
```python
class SingleGame():
    def __init__(self, socket):
        self.socket = socket

    def handle(self, data):
        self.socket.send(data)
```

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

_

_init__

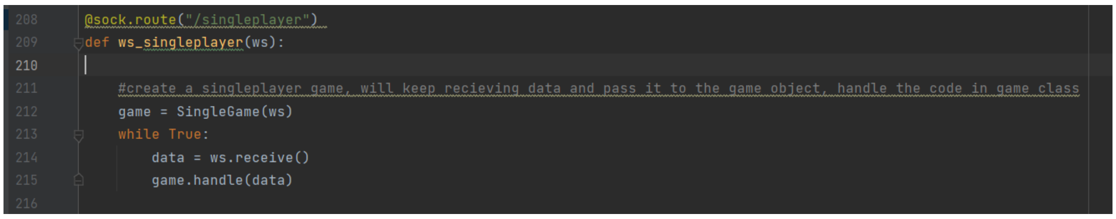
https://github.com/miguelgrinberg/flask-sock/blob/10e0b3bb05580b63433106ef1b5ab42a735846d6/src/flask_sock/__init__.py#L12



```python
    def __init__(self, app=None):
        self.app = None
        self.bp = None
        if app is None:
            self.bp = Blueprint('__flask_sock', __name__)
        else:
            self.app = app
            self.init_app(app)
```

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,

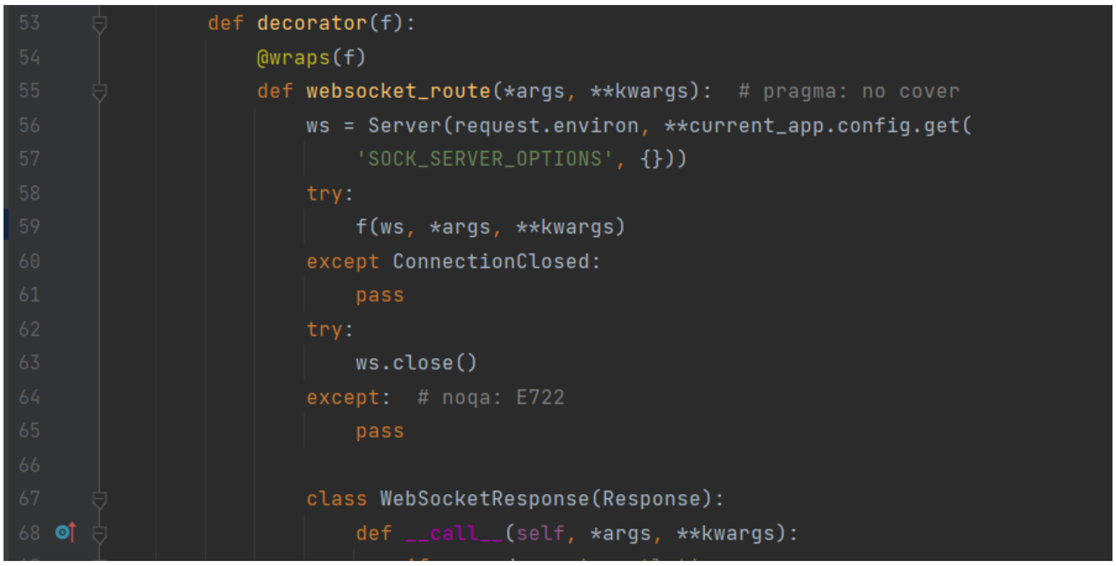


```python
@sock.route("/singleplayer")
def ws_singleplayer(ws):

    #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 **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



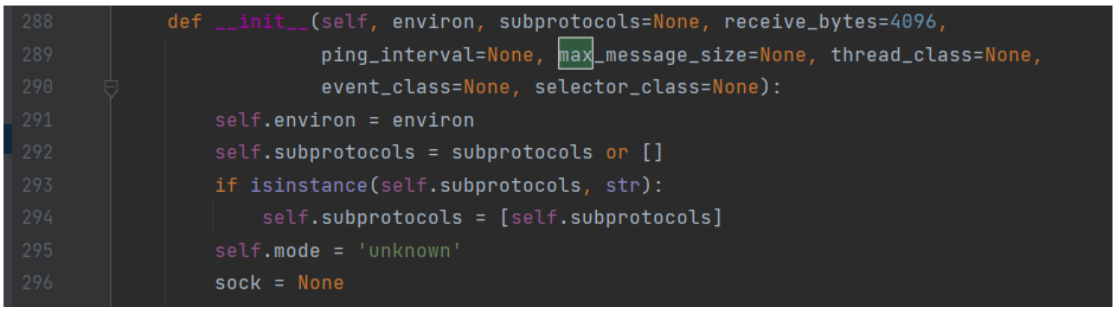
```python
def decorator(f):
    @wraps(f)
    def websocket_route(*args, **kwargs):  # pragma: no cover
        ws = Server(request.environ, **current_app.config.get(
            'SOCK_SERVER_OPTIONS', {}))
        try:
            f(ws, *args, **kwargs)
        except ConnectionClosed:
            pass
        try:
            ws.close()
        except:  # noqa: E722
            pass

        class WebSocketResponse(Response):
            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



```
    def __init__(self, environ, subprotocols=None, receive_bytes=4096,
                 ping_interval=None, max_message_size=None, thread_class=None,
                 event_class=None, selector_class=None):
        self.environ = environ
        self.subprotocols = subprotocols or []
        if isinstance(self.subprotocols, str):
            self.subprotocols = [self.subprotocols]
        self.mode = 'unknown'
        sock = None
```

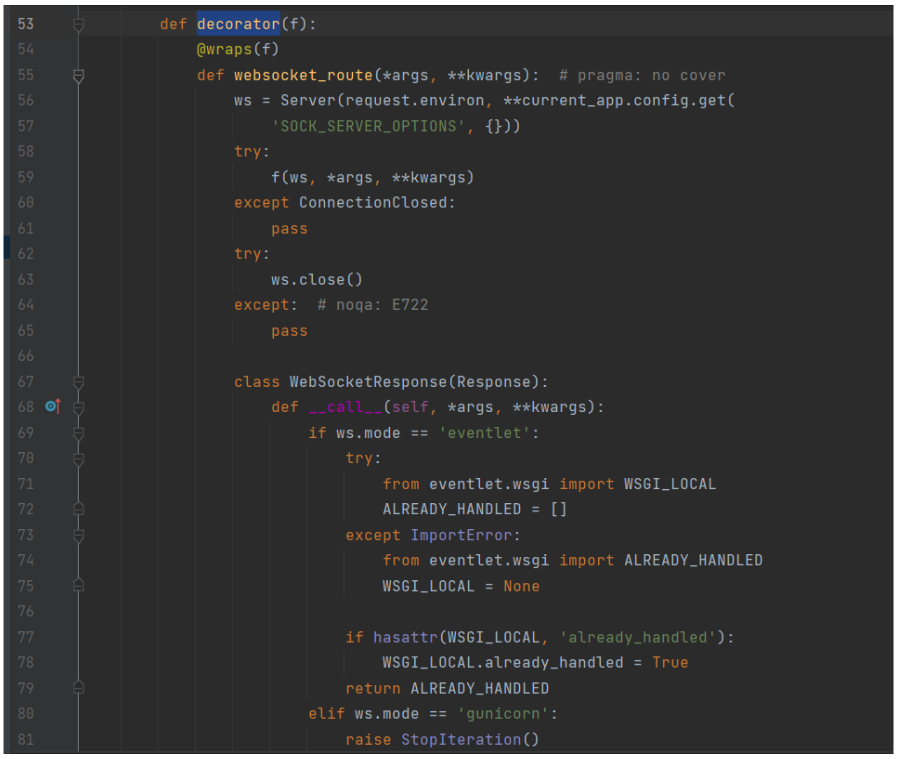
For those different parameters, **(environ， subprotocols， receive_bytes， ping_interval， max_message_size， thread_class， event_class， selector_class).**

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

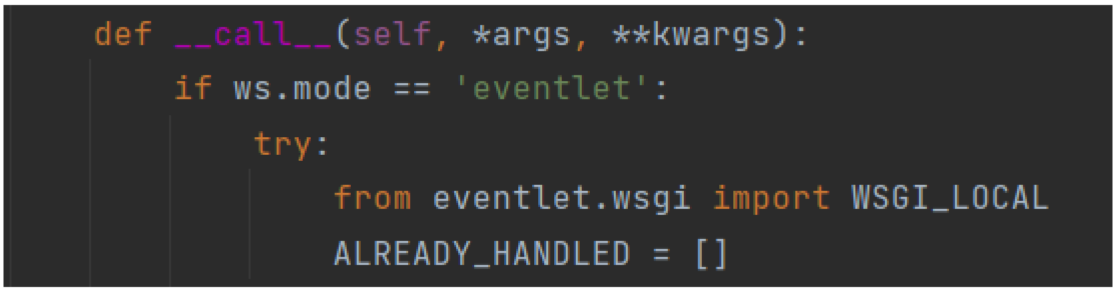


```
        def decorator(f):
            @wraps(f)
            def websocket_route(*args, **kwargs):  # pragma: no cover
                ws = Server(request.environ, **current_app.config.get(
                    'SOCK_SERVER_OPTIONS', {}))
                try:
                    f(ws, *args, **kwargs)
                except ConnectionClosed:
                    pass
                try:
                    ws.close()
                except:  # noqa: E722
                    pass

                class WebSocketResponse(Response):
                    def __call__(self, *args, **kwargs):
                        if ws.mode == 'eventlet':
                            try:
                                from eventlet.wsgi import WSGI_LOCAL
                                ALREADY_HANDLED = []
                            except ImportError:
                                from eventlet.wsgi import ALREADY_HANDLED
                                WSGI_LOCAL = None

                            if hasattr(WSGI_LOCAL, 'already_handled'):
                                WSGI_LOCAL.already_handled = True
                            return ALREADY_HANDLED
                        elif ws.mode == 'gunicorn':
                            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.