[Flask-SocketIO]

General Information & Licensing

Code Repository	https://github.com/miguelgrinberg/Flask-SocketIO
License Type	MIT License
License Description	● Allows for commercial use ● Allows for modification ● Allows for distribution ● Allows for private use
License Restrictions	● There is no warranty of any kind ● The creators are not liable for any kind of damage

[Python-SocketIO]

General Information & Licensing

Code Repository	https://github.com/miguelgrinberg/python-socketio
License Type	MIT License
License Description	● Allows for commercial use ● Allows for modification ● Allows for distribution ● Allows for private use
License Restrictions	● There is no warranty of any kind ● The creators are not liable for any kind of damage

[Python-EngineIO]

General Information & Licensing

Code Repository	https://github.com/miguelgrinberg/python-engineio
License Type	MIT License
License Description	● Allows for commercial use ● Allows for modification ● Allows for distribution

	- Allows for private use
License Restrictions	- There is no warranty of any kind - The creators are not liable for any kind of damage

[Gevent-Websocket]

General Information & Licensing

Code Repository	https://gitlab.com/noppo/gevent-websocket
License Type	Apache License 2.0
License Description	- Allows for commercial use - Allows for modification - Allows for distribution - Allows for private use - Allows for patent use
License Restrictions	- There is no warranty of any kind - The creators are not liable for any kind of damage - There is no trademark use
License Conditions	To use this license, the user must do the following: - The license and copyright notices must be included in any distribution of the code - If any changes are made from the original code, this must be documented.

[Gevent]

General Information & Licensing

Code Repository	https://github.com/gevent/gevent
License Type	MIT License
License Description	- Allows for commercial use - Allows for modification - Allows for distribution - Allows for private use
License Restrictions	- There is no warranty of any kind - The creators are not liable for any kind of damage

Magic ⋆★.°•.°☽˚⁀➷｡˚★彡⋆༄

Flask-SocketIO is an extension of Flask for Python which implements SocketIO for flask servers. SocketIO is a library meant to allow real-time bi-directional communication between a client and a server.

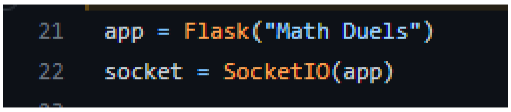
SocketIO's server side code was developed for Node.js, which is not the language we're using for our server. Flask-SocketIO allows us to use it for our Python server. It is meant specifically for servers, and allows for the client to use any version of socketIO for any language to connect to our flask server.

Flask-SocketIO is used in our App.py file, and in there it is used to provide fast communication between the user and our server. It allows for an user to join a room, all users of that room to receive a problem, and for the server to validate and send back the results of each user's answer to the problem.

Setting up things for our server

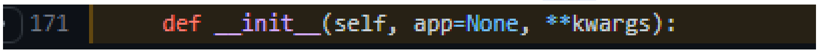
It initializes socketio, and it passes along our flask server details as well

Our code:



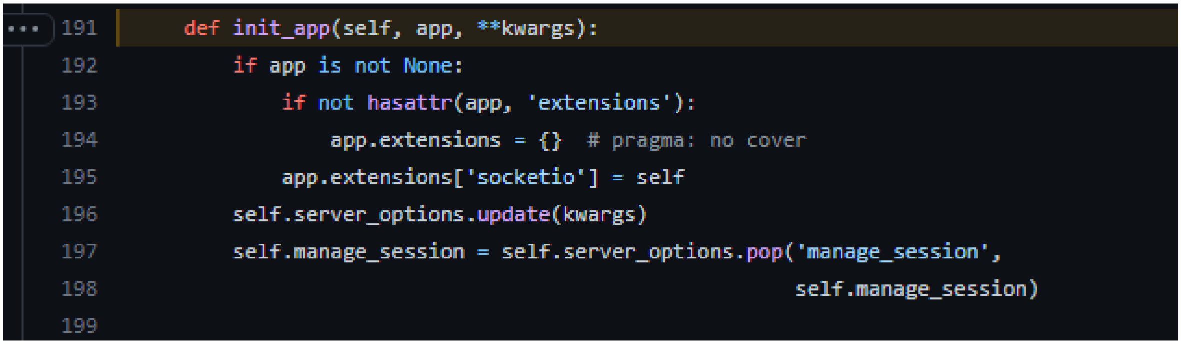
```python
app = Flask("Math Duels")
socket = SocketIO(app)
```

This calls this function in Flask-SocketIO: Link



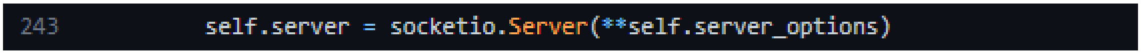
```python
    def __init__(self, app=None, **kwargs):
```

This then calls the self.init_app() function: Link



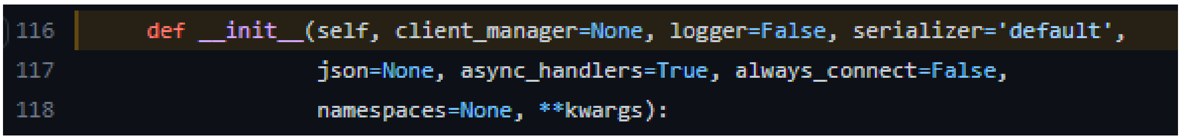
```python
    def init_app(self, app, **kwargs):
        if app is not None:
            if not hasattr(app, 'extensions'):
                app.extensions = {}  # pragma: no cover
            app.extensions['socketio'] = self
        self.server_options.update(kwargs)
        self.manage_session = self.server_options.pop('manage_session',
                                                      self.manage_session)

```

This function then calls upon an external library (python-socketio): Link



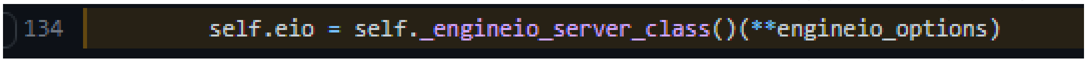
```python
        self.server = socketio.Server(**self.server_options)
```

This external library is initialized by the line of code above. It calls the python-socketio's initialize function: Link



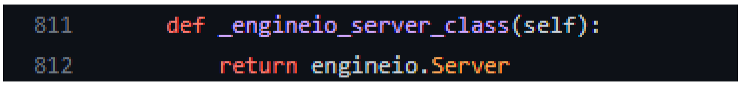
```python
    def __init__(self, client_manager=None, logger=False, serializer='default',
                 json=None, async_handlers=True, always_connect=False,
                 namespaces=None, **kwargs):
```

This code eventually comes to calling another external library (python-engineio): Link



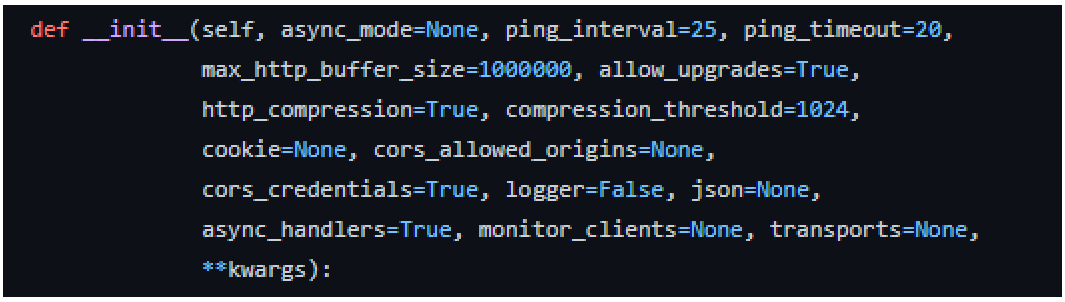
```python
        self.eio = self._engineio_server_class()(**engineio_options)
```

This function calls another function, which is the one actually calling the external library: Link



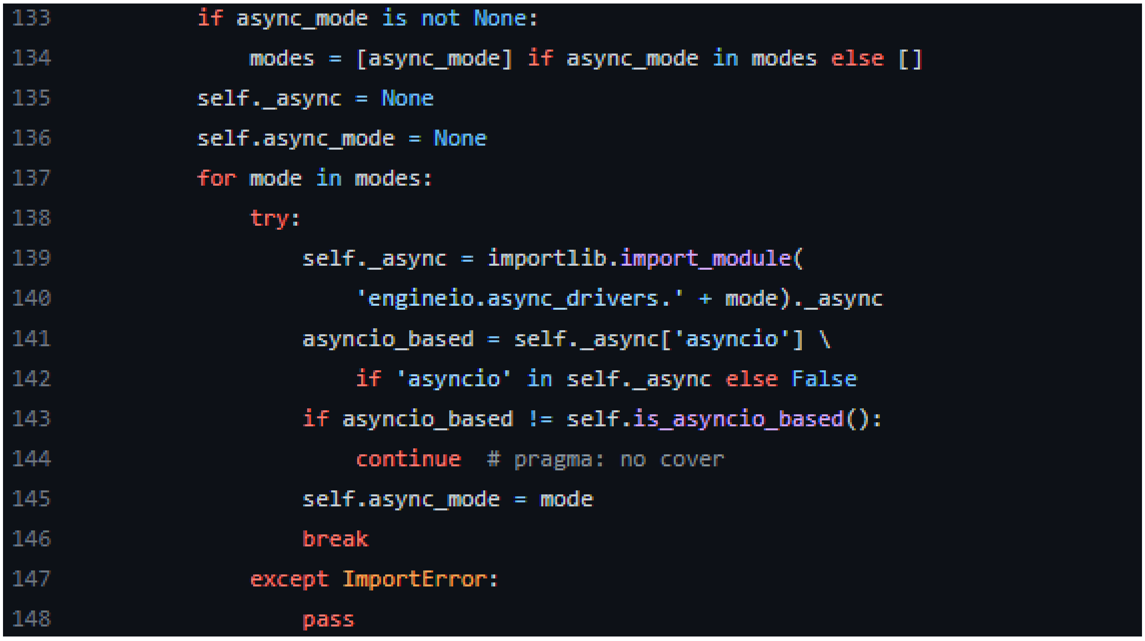
```python
    def _engineio_server_class(self):
        return engineio.Server
```

This external library is initialized by the line of code above. It calls the python-engineio's initialize function. Link



```python
def __init__(self, async_mode=None, ping_interval=25, ping_timeout=20,
             max_http_buffer_size=1000000, allow_upgrades=True,
             http_compression=True, compression_threshold=1024,
             cookie=None, cors_allowed_origins=None,
             cors_credentials=True, logger=False, json=None,
             async_handlers=True, monitor_clients=None, transports=None,
             **kwargs):
```

This eventually leads to this portion of code: Link



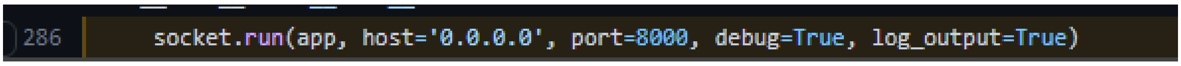
```python
        if async_mode is not None:
            modes = [async_mode] if async_mode in modes else []
        self._async = None
        self.async_mode = None
        for mode in modes:
            try:
                self._async = importlib.import_module(
                    'engineio.async_drivers.' + mode)._async
                asyncio_based = self._async['asyncio'] \
                    if 'asyncio' in self._async else False
                if asyncio_based != self.is_asyncio_based():
                    continue  # pragma: no cover
                self.async_mode = mode
                break
            except ImportError:
                pass
```

What this portion of code does is look for acceptable replacements to use instead of 'long-polling,' which this code will do if it doesn't find any. In our case, we are using gevent-websocket, which this code accepts.

Whenever default SocketIO commands are run, whenever it finds its way down to the EngineIO library, it will redirect it to gevent-websocket instead.

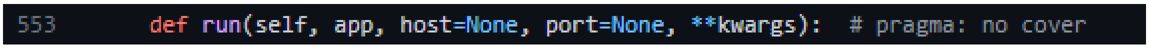
Initial running of server

This runs the server that will handle both WebSockets and regular HTTP requests.
Our Code



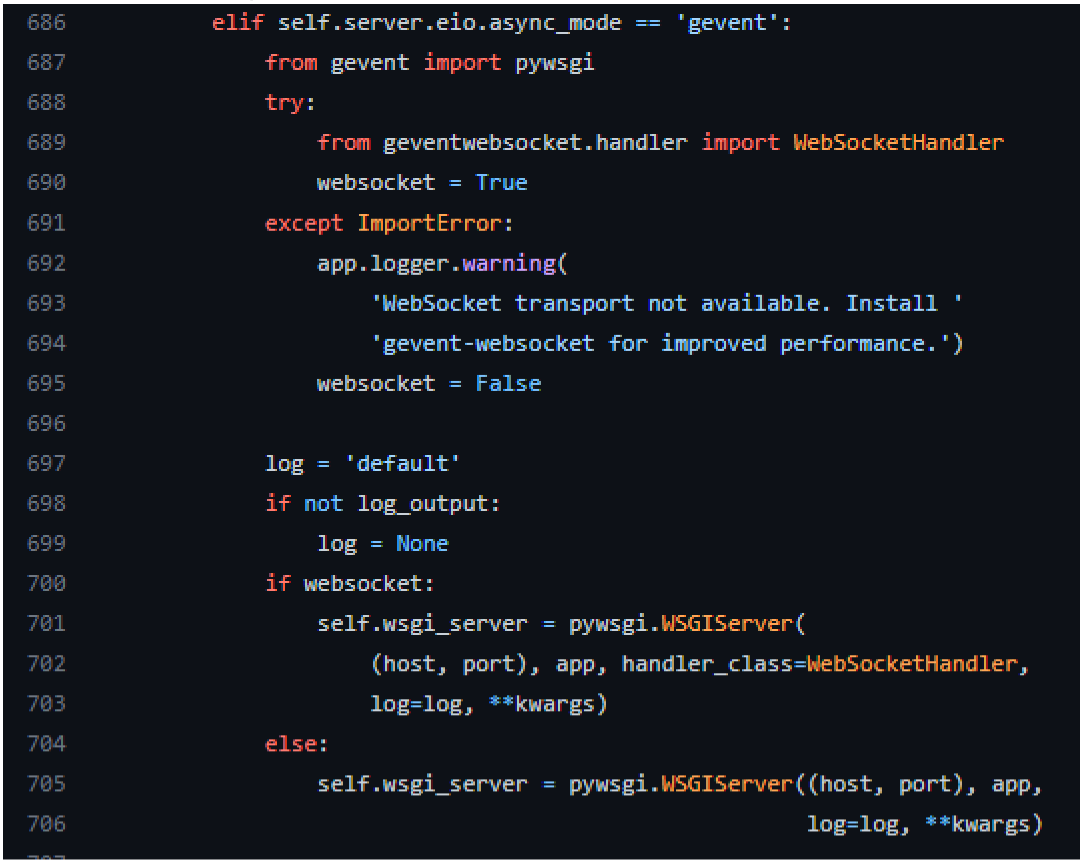
```
        socket.run(app, host='0.0.0.0', port=8000, debug=True, log_output=True)
```

This calls this function in Flask-SocketIO: link



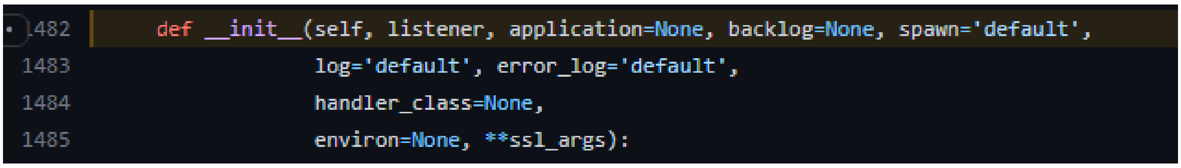
```
    def run(self, app, host=None, port=None, **kwargs):  # pragma: no cover
```

This then calls this portion of code: link



```
686         elif self.server.eio.async_mode == 'gevent':
687             from gevent import pywsgi
688             try:
689                 from geventwebsocket.handler import WebSocketHandler
690                 websocket = True
691             except ImportError:
692                 app.logger.warning(
693                     'WebSocket transport not available. Install '
694                     'gevent-websocket for improved performance.')
695                 websocket = False
696
697             log = 'default'
698             if not log_output:
699                 log = None
700             if websocket:
701                 self.wsgi_server = pywsgi.WSGIServer(
702                     (host, port), app, handler_class=WebSocketHandler,
703                     log=log, **kwargs)
704             else:
705                 self.wsgi_server = pywsgi.WSGIServer((host, port), app,
706                                                      log=log, **kwargs)
```

It does this because we installed gevent-websocket, and set eio.async_mode to ‘gevent’.
This then calls this line 701 of the image linked above.
This calls this function: link

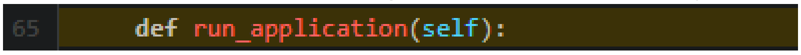


```
    def __init__(self, listener, application=None, backlog=None, spawn='default',
                 log='default', error_log='default',
                 handler_class=None,
                 environ=None, **ssl_args):
```

This has in fact created everything we need to actually begin to use websockets now.

Upgrading to websocket

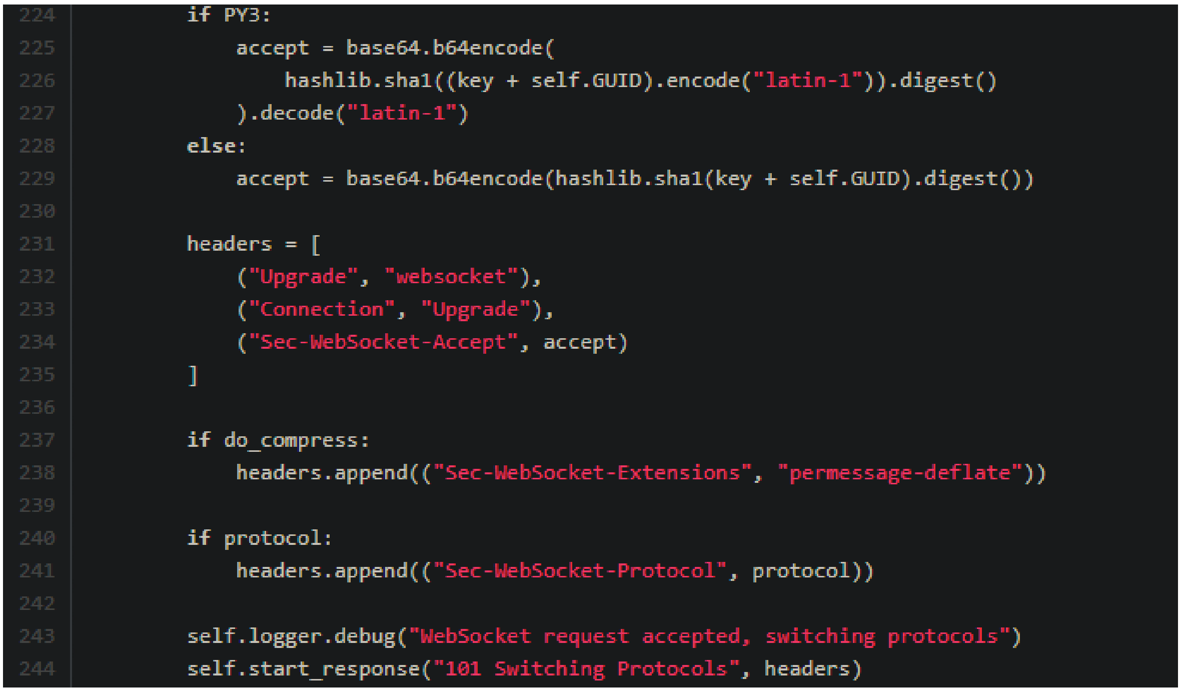
Whenever our user’s join a lobby, they create a websocket.
This calls this function on the gevent-websocket library: link



```
    def run_application(self):
```

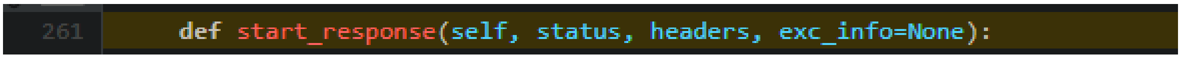
This then calls the upgrade_websocket function, which then calls the upgrade_connection function. upgrade_websocket
upgrade_connection

This is where the Websocket upgrade connection is actually built.
It writes all the headers and stores them.



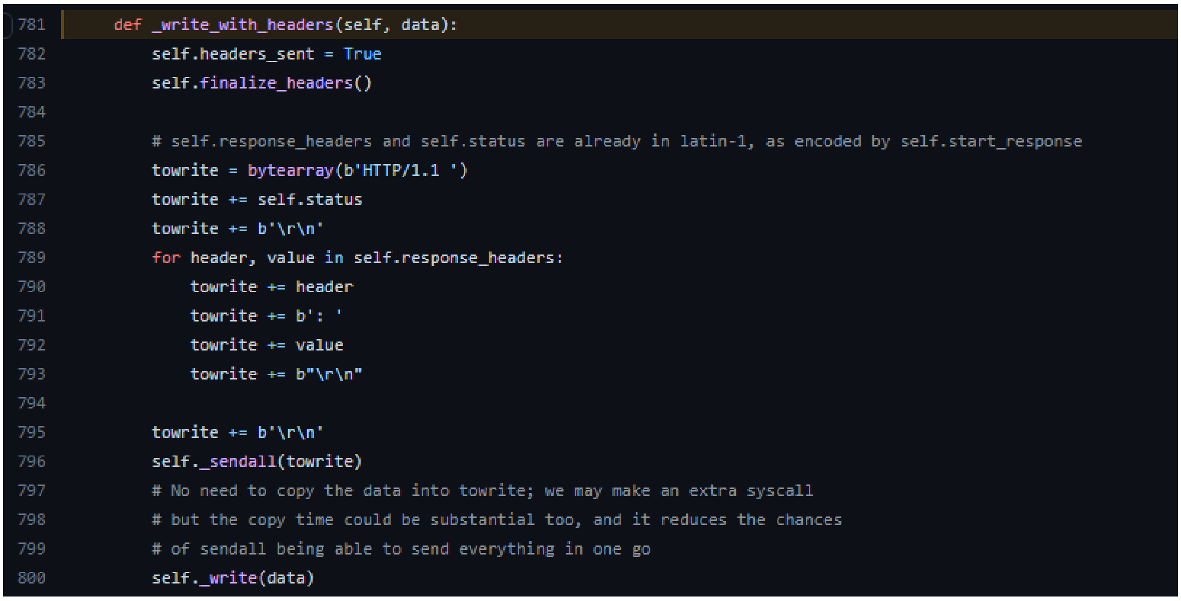
```
        if PY3:
            accept = base64.b64encode(
                hashlib.sha1((key + self.GUID).encode("latin-1")).digest()
            ).decode("latin-1")
        else:
            accept = base64.b64encode(hashlib.sha1(key + self.GUID).digest())

        headers = [
            ("Upgrade", "websocket"),
            ("Connection", "Upgrade"),
            ("Sec-WebSocket-Accept", accept)
        ]

        if do_compress:
            headers.append(("Sec-WebSocket-Extensions", "permessage-deflate"))

        if protocol:
            headers.append(("Sec-WebSocket-Protocol", protocol))

        self.logger.debug("WebSocket request accepted, switching protocols")
        self.start_response("101 Switching Protocols", headers)
```

It then calls the start_response function: [link]()



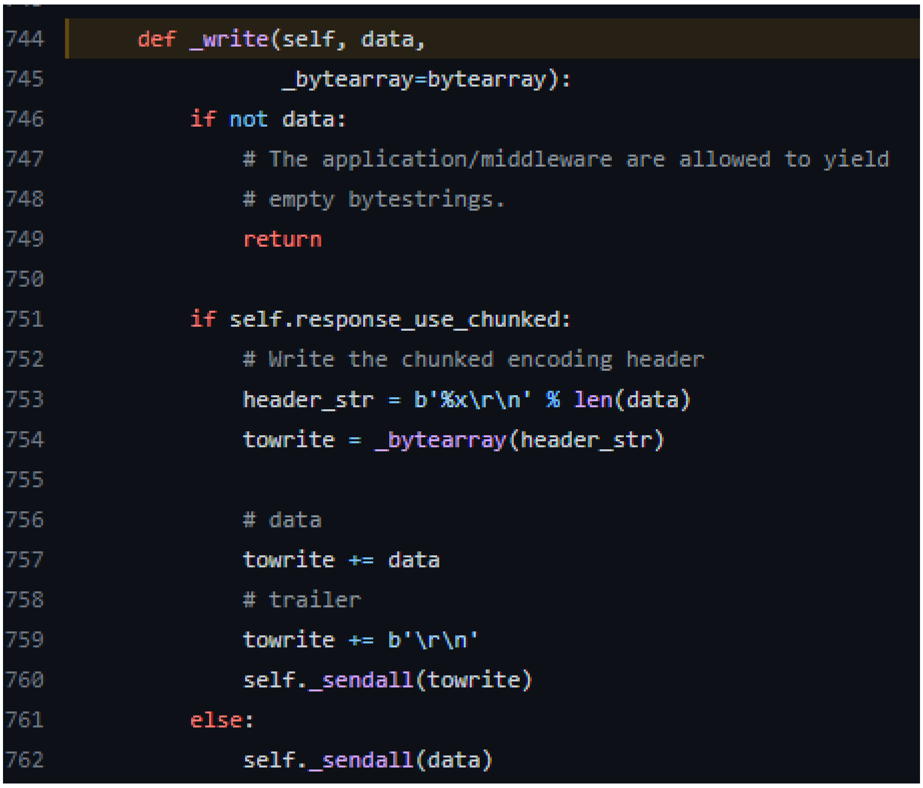
```
    def start_response(self, status, headers, exc_info=None):
```

which calls gevent's start_response function. This then calls it's write function, which then calls it's _write_with_headers function: [link]()



```
    def _write_with_headers(self, data):
        self.headers_sent = True
        self.finalize_headers()

        # self.response_headers and self.status are already in latin-1, as encoded by self.start_response
        towrite = bytearray(b'HTTP/1.1 ')
        towrite += self.status
        towrite += b'\r\n'
        for header, value in self.response_headers:
            towrite += header
            towrite += b': '
            towrite += value
            towrite += b"\r\n"

        towrite += b'\r\n'
        self._sendall(towrite)
        # No need to copy the data into towrite; we may make an extra syscall
        # but the copy time could be substantial too, and it reduces the chances
        # of sendall being able to send everything in one go
        self._write(data)
```

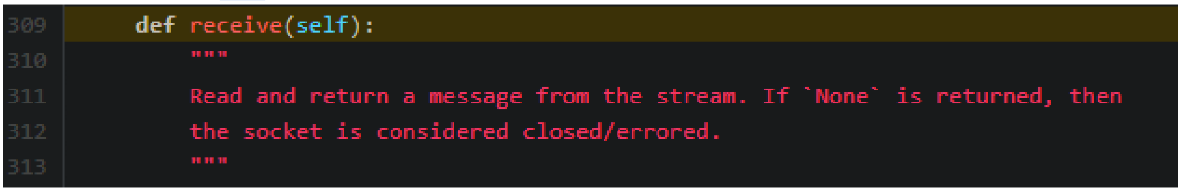
This then calls _write which encodes it and sends it. link



```
    def _write(self, data,
               _bytearray=bytearray):
        if not data:
            # The application/middleware are allowed to yield
            # empty bytestrings.
            return

        if self.response_use_chunked:
            # Write the chunked encoding header
            header_str = b'%x\r\n' % len(data)
            towrite = _bytearray(header_str)

            # data
            towrite += data
            # trailer
            towrite += b'\r\n'
            self._sendall(towrite)
        else:
            self._sendall(data)
```

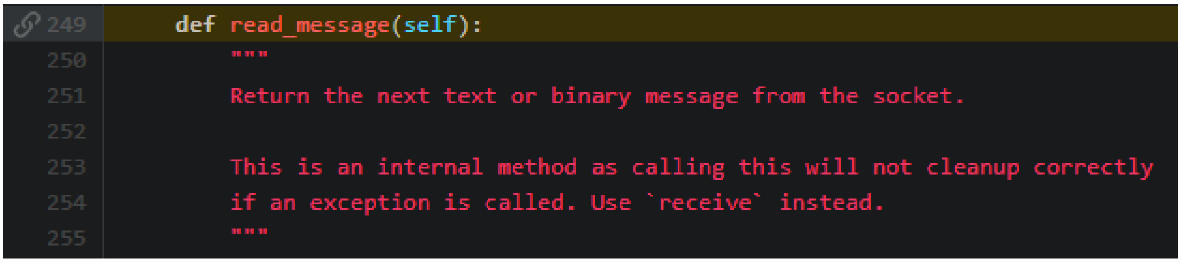
User sending data

When a user sends data through the websocket connection, this call's gevent-websocket receive function: link



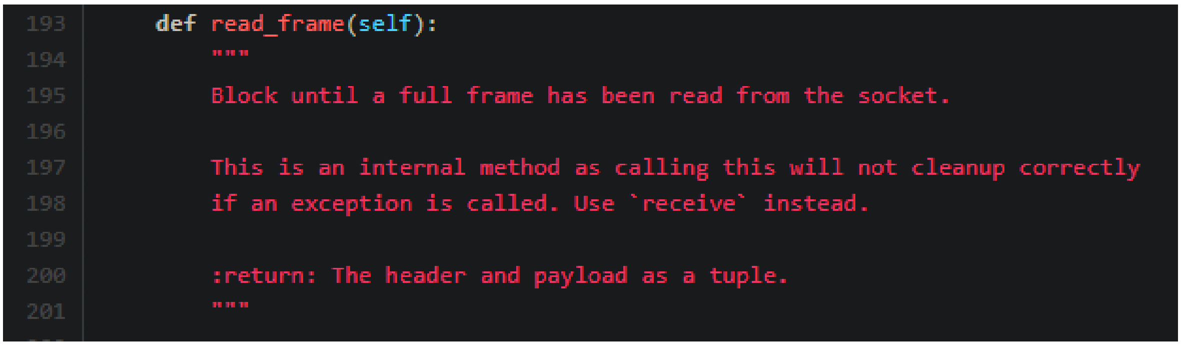
```
    def receive(self):
        """
        Read and return a message from the stream. If `None` is returned, then
        the socket is considered closed/errored.
        """
```

This function then calls an internal function called read_message: link



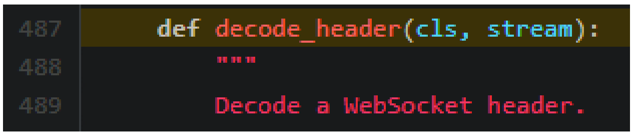
```
    def read_message(self):
        """
        Return the next text or binary message from the socket.

        This is an internal method as calling this will not cleanup correctly
        if an exception is called. Use `receive` instead.
        """
```

This function calls read_frame, and ends it immediately if it is a ping, pong or close request. Otherwise, it repeats calling read_frame until it finishes reading all data. link to read_frame



```python
    def read_frame(self):
        """
        Block until a full frame has been read from the socket.

        This is an internal method as calling this will not cleanup correctly
        if an exception is called. Use `receive` instead.

        :return: The header and payload as a tuple.
        """
```

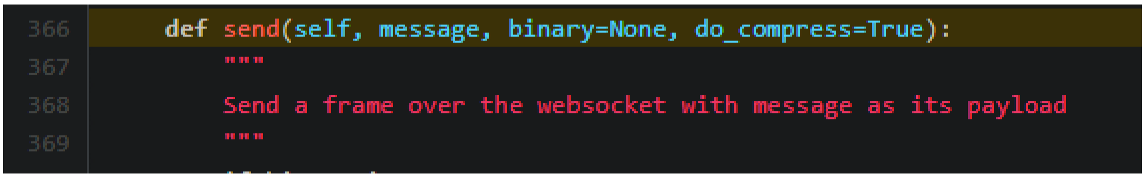
Every time read_frame is called, it will also call decode_headers. This function finds the headers in the websocket frame, and parses them to make them easily available to the rest of the code. link



```python
    def decode_header(cls, stream):
        """
        Decode a WebSocket header.
```

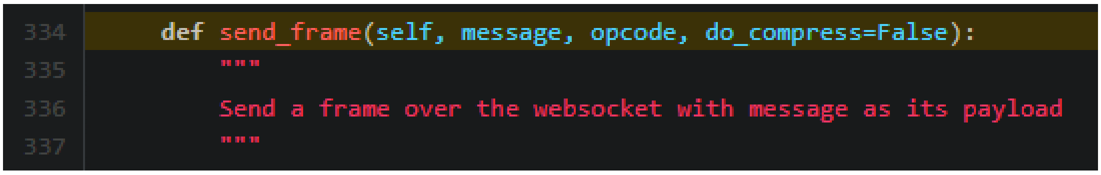
User receiving data

When we want to send data from the server, the send function is called. link



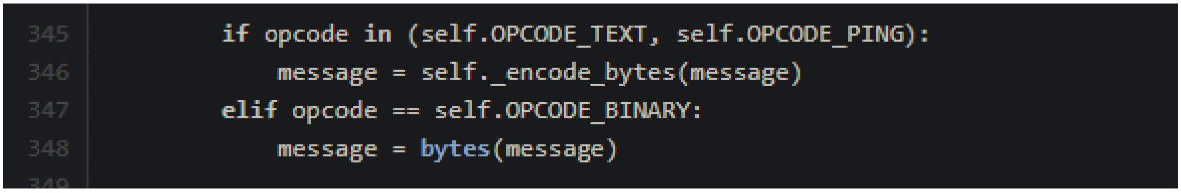
```python
    def send(self, message, binary=None, do_compress=True):
        """
        Send a frame over the websocket with message as its payload
        """
```

This calls the send_frame function. link

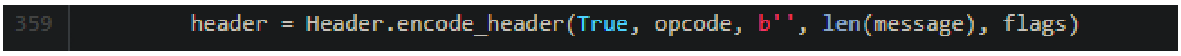


```python
    def send_frame(self, message, opcode, do_compress=False):
        """
        Send a frame over the websocket with message as its payload
        """
```

This encodes the data properly if it is text, ping or binary.

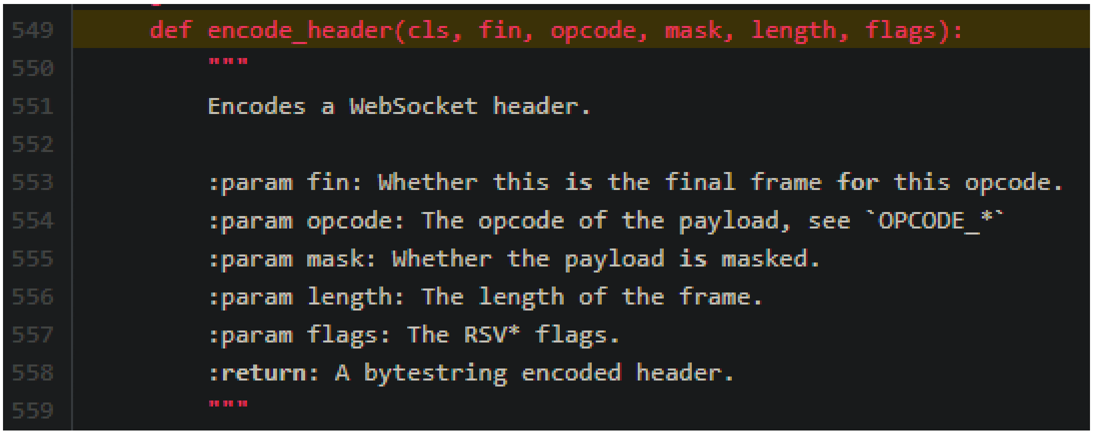


```python
        if opcode in (self.OPCODE_TEXT, self.OPCODE_PING):
            message = self._encode_bytes(message)
        elif opcode == self.OPCODE_BINARY:
            message = bytes(message)
```



```python
        header = Header.encode_header(True, opcode, b'', len(message), flags)
```

For all types of data however, the encode_header function is called. In this instance with parameters to ensure the client understands what it's being sent. link to encode_header

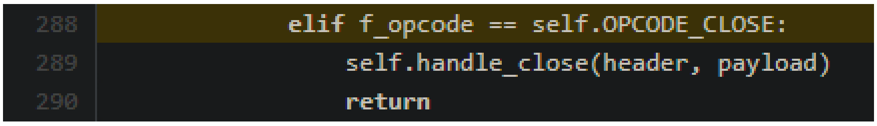


```python
    def encode_header(cls, fin, opcode, mask, length, flags):
        """
        Encodes a WebSocket header.

        :param fin: Whether this is the final frame for this opcode.
        :param opcode: The opcode of the payload, see `OPCODE_*`
        :param mask: Whether the payload is masked.
        :param length: The length of the frame.
        :param flags: The RSV* flags.
        :return: A bytestring encoded header.
        """
```

This function returns all the headers back to send_frame, which sends that + the encoded payload back to the user.

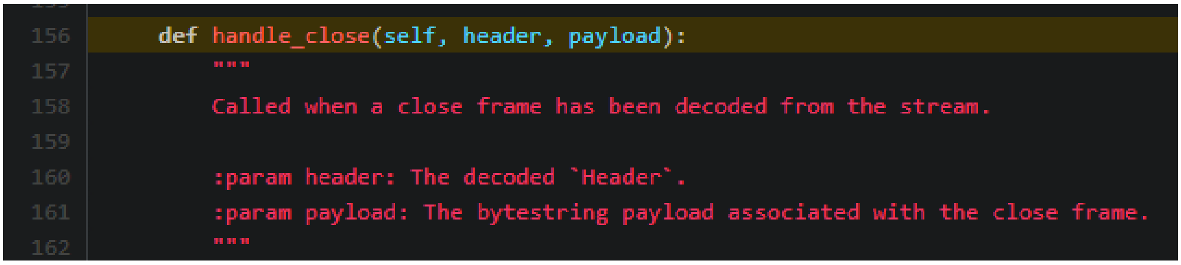
User closing websocket

If the read_message function detects an Opcode close, it will call the function handle_close. link



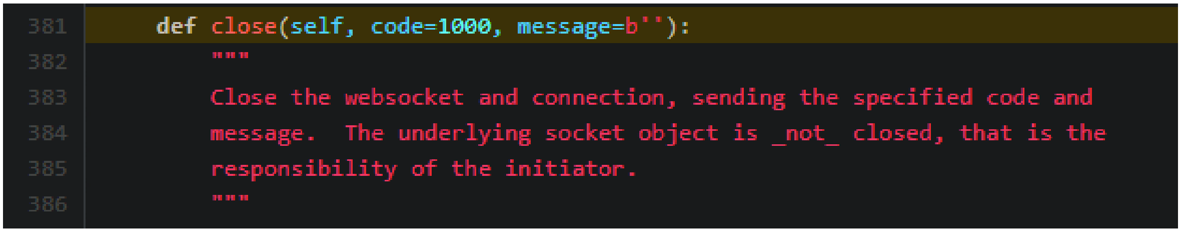
```
        elif f_opcode == self.OPCODE_CLOSE:
            self.handle_close(header, payload)
            return
```

link to handle_close



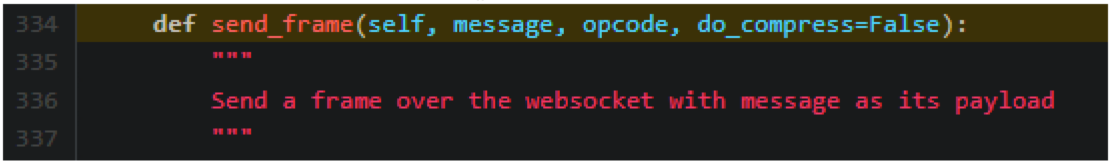
```
    def handle_close(self, header, payload):
        """
        Called when a close frame has been decoded from the stream.

        :param header: The decoded `Header`.
        :param payload: The bytestring payload associated with the close frame.
        """
```

This function then calls close(). link



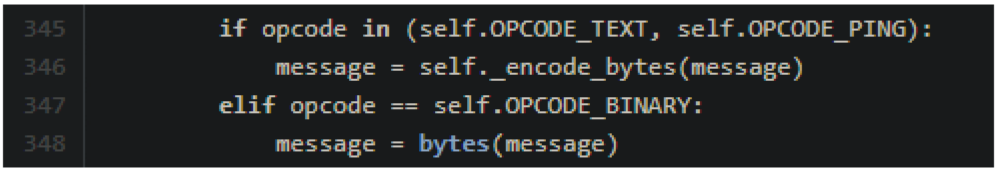
```
    def close(self, code=1000, message=b''):
        """
        Close the websocket and connection, sending the specified code and
        message.  The underlying socket object is _not_ closed, that is the
        responsibility of the initiator.
        """
```

It calls the function _encode_bytes to encode its message.
It then calls the function send_frame(): link



```
    def send_frame(self, message, opcode, do_compress=False):
        """
        Send a frame over the websocket with message as its payload
        """
```

As the opcode is not text, binary or ping, it does not re-encode the message, and simply sends that function back to the user. link



```
        if opcode in (self.OPCODE_TEXT, self.OPCODE_PING):
            message = self._encode_bytes(message)
        elif opcode == self.OPCODE_BINARY:
            message = bytes(message)
```