Open-Source Report

[simple-websocket]

General Information & Licensing

Code Repository	https://github.com/miguelgrinberg/simple-websocket
License Type	MIT License
License Description	• Under the MIT license people can freely "use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software" (source)
License Restrictions	• All copies or large sections of the software must include a copyright notice and permissions notice. (source)

Purpose

The simple-websocket library allows us to handle websocket connections between the client and server on the backend when users visit the '/lobby' or '/game'. In particular, we add '/lobby-websocket' and '/game-websocket' endpoints to app.route in app.py with a websocket=True argument to handle upgrade requests at these paths. We then create an instance of the websocket class by calling simple_websocket.Server(request.environ) and call the receive method of this class to get more data or the close method to close the connection.

Magic ★★˚｡⋆｡˚☽˚｡⋆🌊˚★

Once app.py receives an upgrade request from the client at either '/lobby-websocket' '/game-websocket', create an instance of the Server() class from the simple_websocket library and pass in the environ attribute of the request object generated by flask. Server() will do the following to generate response to the handshake request received by the client to establish a websocket connection:

- The constructor of the inherited Base class is called here.
- Base calls self.handshake() in its constructor here.
- handshake() calls self._handle_events() here.
- To generate the response to the incoming upgrade request, self._handle_events() calls the send() method of ws which is an instance of the WSConnection class here.
- The send() method of WSConnection imported from the wsproto library generates the response data by calling the send() method of the H11Handshake class here.
- Since event is an instance of AcceptConnection, H11Handshake.send() calls self._accept() here.
- H11Handshake._accept() generates the Sec-Websocket-Accept token by calling generate_accept_token() function on the sec-websocket-key value from the request header here.
- generate_accept_token() appends the GUID to the sec-websocket-key value here, then computes the SHA-1 hash and base64 encoding of this here and here.
- Once the accept token is generated by generate_accept_token(), H11Handshake._accept() inserts the token into the response header here.
- H11Handshake._accept() then generates the full http response here and returns it here.
- H11Handshake.send() also returns the generated response byte array here, as does WSConneciton.send() here.
- out_data therefore gets assigned to the full byte array response in _handle_events() here which then gets sent over a TCP connection to the client here.

Once a connection between the server and client has been established as a result of the call to self.handshake() in the constructor of the Base class in the simple_websocket library, the constructor continues by beginning a new thread here that uses self._thread() to handle incoming frames. self._thread() calls self.sock.recv() to receive incoming frames from the socket here. Incoming frames then handled the following way:

- self._handle_events() is called here.
- Within _handle_events(), the events() method from ws which is an instance of the WSConnection is called here.
- WSConnection.events() then calls the events() method of the Connection class here.
- Connection.events() then calls received_frames() method from the FrameProtocol class here.
- FrameProtocol.received_frames() then calls FrameProtocol._parse_more_gen() (since self._parse_more references this method) here.
- FrameProtocol._parse_more_gen() calls the process_buffer() method of the FrameDecoder class here.
- FrameDecoder.process_buffer() calls self.parse_header() here.
- FrameDecoder.parse_header() then retrieves the first two bytes of the frame here then determines the value of the frame FIN bit by the AND of value of the very first byte of the frame FIN_MASK which is 0x80 here.
  - The rsv bits are determined by additional masking on the first byte here.

    - The opcode is determined by masking the first byte with OPCODE_MASK which is 0x0f here. The meaning of the opcode is determined by calling the Opcode class here.
    - Whether or not a mask is used is determined by masking the second byte of the frame with MASK_MASK which is 0x80 here.
    - The value of the first 7 bits of the the payload length is determined by masking the second frame byte with PAYLOAD_LEN_MASK which is 0x7f here.
    - self.parse_extended_payload _length() is then called here to determine if there is an extended payload length. It does this by checking if the payload length is equal to 126 here or 127 here.
    - Once self.parse_extended_payload _length() returns the proper length of the layload, the payload mask is found (if a mask is being used) by retrieving the 4 bytes after the payload length here.
- Once FrameDecoder.parse_header()has parsed all of the websocket frame information, FrameDecoder.process_buffer() uses the length of the payload to retrieve the payload from the raw data here.
- To unmask the payload, the process method of the XorMaskerSimple class is called here. XorMaskerSimple.process() uses the parsed mask to unmask the payload here.
- Once the full frame has been parsed, FrameDecoder.process_buffer generates an instance of the Frame class here which contains all the frame information and returns it.
- This enables FrameProtocol._parse_more_gen() to also return the parsed frame here and therefore FrameProtocol.received_frames() returns the parsed frame as an event here.
- Connection.events() and WSConnection.events() then yield the parsed frame information here and here allowing Base._handle_events() to append the payload to self.input_buffer here if the incoming message over the socket is a text message.
- simple_websocket.Server.recieve() then returns the first item from self.input_buffer here. This is the method that we use to read data from the websocket in app.py.

simple_websocket.Server.send() does the following to send create and send a websocket frame:

- WSConnection.send() is called here.
- WSConnection.send() calls Connection.send() here
- Connection.send() calls FrameProtocol.send_data() here.
- FrameProtocol.send_data() generates the opcode here. Then calls self._serialize_frame() here.
- The fin bit (which is 1 by default), rsv, and opcode are combined when _serialize_frame() calls self._make_fin_rsv_opcode() here. self._make_fin_rsv_opcode() shifts and combines the values here.
- The payload length is determined in _serialize_frame() here.
- The header is combined as a byte array here and then is combined with the payload and returned to complete the frame here.
- This FrameProtocol.send_data() also returns this value here as does Connection.send() here and WSConnection.send() here and simple_websocket.Server.send() sends the frame over the websocket here.

The stack trace for simple_websocket.Server.close() starts with a call to WSConnection.send() here then a call to Connection.send() here. FrameProtocal.close() is then called here which calls FrameProtocol._serialize_frame() here which is used in the same way as it was during the execution of simple_websocket.Server.send(). Since FrameProtocal.close() returns the generated frame returned by _serialize_frame() here, so does Connection.send() here and

WSConnection.send() here which allows simple_websocket.Server.close() to send the frame over the websocket here.