Report: Flask-SocketIO

Table of Contents

1. **Introduction** **3**
2. **Websocket Handshakes** **3**
    a. compute_accept_value 4
    b. _accept_connection 5
3. **Websocket Frame Parsing** **6**
    a. Receiving Frames 7
    b. Sending Frames 10
    c. _websocket_mask 11
4. **class SocketIO** **12**
    a. method on 12
    b. method run 13
5. **join_room** **14**
6. **leave_room** **15**
7. **emit** **16**
8. **License** **17**

Report: Flask-SocketIO

The project utilizes the Flask-SocketIO library, the code for which can be found at: https://github.com/miguelgrinberg/Flask-SocketIO/blob/master/flask_socketio/__init__.py

The Flask-SocketIO library gives Flask applications the ability for bidirectional communications between the server and the clients. Flask-SocketIO is built on top of python-socketio (https://github.com/miguelgrinberg/python-socketio), and essentially wraps a Flask application with the socket functionality.

This document also covers how Flask-SocketIO handles websocket handshakes and websocket frame sending & receiving. `Flask-SocketIO` is built on top of `python-socketio`, which further utilises `engineio`, which relies on `tornado` for websocket handshakes and frame sending & receiving. The source code for which can be found here https://github.com/tornadoweb/tornado. `tornado` implements the final version of the WebSocket Protocol as defined in RFC 6455, https://tools.ietf.org/html/rfc6455.

- `Handshakes`

Before a websocket connection is established, `Flask-SocketIO` relies on polling, in order to upgrade the connection to a websocket, a websocket handshake described in RFC 6455 is performed.

The `"Sec-WebSocket-Accept"` value is computed using the following function.

```
def compute_accept_value(key: Union[str, bytes]) -> str:
    """Computes the value for the Sec-WebSocket-Accept header,
    given the value for Sec-WebSocket-Key.
    """
    sha1 = hashlib.sha1()
    sha1.update(utf8(key))
    sha1.update(b"258EAFA5-E914-47DA-95CA-C5AB0DC85B11")  # Magic value
    return native_str(base64.b64encode(sha1.digest()))
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/websocket.py

After performing all sorts of error checking and handling, if the client sends over the correct headers for a websocket upgrade, the `_accept_connection` function is called. The source code from `tornado` for `_accept_connection` is defined below.

```
async def _accept_connection(self, handler: WebSocketHandler) -> None:
    subprotocol_header = handler.request.headers.get("Sec-WebSocket-Protocol")
    if subprotocol_header:
        subprotocols = [s.strip() for s in subprotocol_header.split(",")]
    else:
        subprotocols = []
    self.selected_subprotocol = handler.select_subprotocol(subprotocols)
    if self.selected_subprotocol:
        assert self.selected_subprotocol in subprotocols
        handler.set_header("Sec-WebSocket-Protocol", self.selected_subprotocol)

    extensions = self._parse_extensions_header(handler.request.headers)
    for ext in extensions:
        if ext[0] == "permessage-deflate" and self._compression_options is not None:
            # TODO: negotiate parameters if compression_options
            # specifies limits.
            self._create_compressors("server", ext[1], self._compression_options)
            if (
                "client_max_window_bits" in ext[1]
                and ext[1]["client_max_window_bits"] is None
            ):
                # Don't echo an offered client_max_window_bits
                # parameter with no value.
                del ext[1]["client_max_window_bits"]
            handler.set_header(
                "Sec-WebSocket-Extensions",
                httputil._encode_header("permessage-deflate", ext[1]),
            )
            break

    handler.clear_header("Content-Type")
    handler.set_status(101)
    handler.set_header("Upgrade", "websocket")
    handler.set_header("Connection", "Upgrade")
    handler.set_header("Sec-WebSocket-Accept", self._challenge_response(handler))
    handler.finish()

    self.stream = handler._detach_stream()

    self.start_pinging()
    try:
        open_result = handler.open(*handler.open_args, **handler.open_kwargs)
        if open_result is not None:
            await open_result
    except Exception:
        handler.log_exception(*sys.exc_info())
        self._abort()
        return

    await self._receive_frame_loop()
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/websocket.py

- Frame parsing - Receiving & Sending

In order to send & receive websocket frames, `Flask-SocketIO` relies on `tornado` for this functionality. Below is the source code for how `tornado` helps `Flask-SocketIO` in sending & receiving frames. Tornado uses bitwise operation to parse a frame, it curates & parses frames by byte data parsing, comparisons and manipulation.

Here are some variables defined in the source code that are used throughout the process of sending & receiving frames.

```
# Bit masks for the first byte of a frame.
FIN = 0x80
RSV1 = 0x40
RSV2 = 0x20
RSV3 = 0x10
RSV_MASK = RSV1 | RSV2 | RSV3
OPCODE_MASK = 0x0F
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/websocket.py

- Receiving Frames

The next three screen grabs are the source code for how `tornado` receives frames over a websocket connection and parses them. The last screenshot is a helper function that implements very important functionality.

```python
async def _receive_frame(self) -> None:
    # Read the frame header.
    data = await self._read_bytes(2)
    header, mask_payloadlen = struct.unpack("BB", data)
    is_final_frame = header & self.FIN
    reserved_bits = header & self.RSV_MASK
    opcode = header & self.OPCODE_MASK
    opcode_is_control = opcode & 0x8
    if self._decompressor is not None and opcode != 0:
        # Compression flag is present in the first frame's header,
        # but we can't decompress until we have all the frames of
        # the message.
        self._frame_compressed = bool(reserved_bits & self.RSV1)
        reserved_bits &= ~self.RSV1
    if reserved_bits:
        # client is using as-yet-undefined extensions; abort
        self._abort()
        return
    is_masked = bool(mask_payloadlen & 0x80)
    payloadlen = mask_payloadlen & 0x7F

    # Parse and validate the length.
    if opcode_is_control and payloadlen >= 126:
        # control frames must have payload < 126
        self._abort()
        return
    if payloadlen < 126:
        self._frame_length = payloadlen
    elif payloadlen == 126:
        data = await self._read_bytes(2)
        payloadlen = struct.unpack("!H", data)[0]
    elif payloadlen == 127:
        data = await self._read_bytes(8)
        payloadlen = struct.unpack("!Q", data)[0]
    new_len = payloadlen
    if self._fragmented_message_buffer is not None:
        new_len += len(self._fragmented_message_buffer)
    if new_len > self.params.max_message_size:
        self.close(1009, "message too big")
        self._abort()
        return
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/websocket.py

```
# Read the payload, unmasking if necessary.
if is_masked:
    self._frame_mask = await self._read_bytes(4)
data = await self._read_bytes(payloadlen)
if is_masked:
    assert self._frame_mask is not None
    data = _websocket_mask(self._frame_mask, data)

# Decide what to do with this frame.
if opcode_is_control:
    # control frames may be interleaved with a series of fragmented
    # data frames, so control frames must not interact with
    # self._fragmented_*
    if not is_final_frame:
        # control frames must not be fragmented
        self._abort()
        return
elif opcode == 0:  # continuation frame
    if self._fragmented_message_buffer is None:
        # nothing to continue
        self._abort()
        return
    self._fragmented_message_buffer += data
    if is_final_frame:
        opcode = self._fragmented_message_opcode
        data = self._fragmented_message_buffer
        self._fragmented_message_buffer = None
else:  # start of new data message
    if self._fragmented_message_buffer is not None:
        # can't start new message until the old one is finished
        self._abort()
        return
    if not is_final_frame:
        self._fragmented_message_opcode = opcode
        self._fragmented_message_buffer = data

if is_final_frame:
    handled_future = self._handle_message(opcode, data)
    if handled_future is not None:
        await handled_future
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/websocket.py

```
def _handle_message(self, opcode: int, data: bytes) -> "Optional[Future[None]]":
    """Execute on_message, returning its Future if it is a coroutine."""
    if self.client_terminated:
        return None

    if self._frame_compressed:
        assert self._decompressor is not None
        try:
            data = self._decompressor.decompress(data)
        except _DecompressTooLargeError:
            self.close(1009, "message too big after decompression")
            self._abort()
            return None

    if opcode == 0x1:
        # UTF-8 data
        self._message_bytes_in += len(data)
        try:
            decoded = data.decode("utf-8")
        except UnicodeDecodeError:
            self._abort()
            return None
        return self._run_callback(self.handler.on_message, decoded)
    elif opcode == 0x2:
        # Binary data
        self._message_bytes_in += len(data)
        return self._run_callback(self.handler.on_message, data)
    elif opcode == 0x8:
        # Close
        self.client_terminated = True
        if len(data) >= 2:
            self.close_code = struct.unpack(">H", data[:2])[0]
        if len(data) > 2:
            self.close_reason = to_unicode(data[2:])
        # Echo the received close code, if any (RFC 6455 section 5.5.1).
        self.close(self.close_code)
    elif opcode == 0x9:
        # Ping
        try:
            self._write_frame(True, 0xA, data)
        except StreamClosedError:
            self._abort()
        self._run_callback(self.handler.on_ping, data)
    elif opcode == 0xA:
        # Pong
        self.last_pong = IOLoop.current().time()
        return self._run_callback(self.handler.on_pong, data)
    else:
        self._abort()
    return None
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/websocket.py

- Sending Frames

Below is the source code for how `tornado` curates and sends frames over a websocket connection.

```
def _write_frame(
    self, fin: bool, opcode: int, data: bytes, flags: int = 0
) -> "Future[None]":
    data_len = len(data)
    if opcode & 0x8:
        # All control frames MUST have a payload length of 125
        # bytes or less and MUST NOT be fragmented.
        if not fin:
            raise ValueError("control frames may not be fragmented")
        if data_len > 125:
            raise ValueError("control frame payloads may not exceed 125 bytes")
    if fin:
        finbit = self.FIN
    else:
        finbit = 0
    frame = struct.pack("B", finbit | opcode | flags)
    if self.mask_outgoing:
        mask_bit = 0x80
    else:
        mask_bit = 0
    if data_len < 126:
        frame += struct.pack("B", data_len | mask_bit)
    elif data_len <= 0xFFFF:
        frame += struct.pack("!BH", 126 | mask_bit, data_len)
    else:
        frame += struct.pack("!BQ", 127 | mask_bit, data_len)
    if self.mask_outgoing:
        mask = os.urandom(4)
        data = mask + _websocket_mask(mask, data)
    frame += data
    self._wire_bytes_out += len(frame)
    return self.stream.write(frame)
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/websocket.py

The source code for function `_websocket_mask`, that is utilised for both sending and receiving frames, can be found below. The function

```python
def _websocket_mask_python(mask: bytes, data: bytes) -> bytes:
    """Websocket masking function.

    `mask` is a `bytes` object of length 4; `data` is a `bytes` object of any length.
    Returns a `bytes` object of the same length as `data` with the mask applied
    as specified in section 5.3 of RFC 6455.

    This pure-python implementation may be replaced by an optimized version when available.
    """
    mask_arr = array.array("B", mask)
    unmasked_arr = array.array("B", data)
    for i in range(len(data)):
        unmasked_arr[i] = unmasked_arr[i] ^ mask_arr[i % 4]
    return unmasked_arr.tobytes()
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/util.py

As seen in the snapshot below `_websocket_mask` in an alias for the function `_websocket_mask_python`.

```python
if os.environ.get("TORNADO_NO_EXTENSION") or os.environ.get("TORNADO_EXTENSION") == "0":
    # These environment variables exist to make it easier to do performance
    # comparisons; they are not guaranteed to remain supported in the future.
    _websocket_mask = _websocket_mask_python
else:
    try:
        from tornado.speedups import websocket_mask as _websocket_mask
    except ImportError:
        if os.environ.get("TORNADO_EXTENSION") == "1":
            raise
        _websocket_mask = _websocket_mask_python
```

This snapshot is from the following source code:
https://github.com/tornadoweb/tornado/blob/master/tornado/util.py

More specifically the project utilizes four specific functions from Flask_SocketIO library, `class SocketIO(on,run), join_room, leave_room & emit`. A detailed summary of how the functions work can be found below.

Note: There are some references to line numbers, these refer to line numbers to the linked file that can be accessed by the link provided above.

- `class SocketIO(Line#56)`

What does this technology accomplish for us?

This class creates a Flask-SocketIO server, which accepts a Flask application instance as a parameter and initialises data structures and variables to hold the different parameters and instance handlers. When the application initialises the socket server by calling `SocketIO(app)`, this is when the Flask application instance is wrapped in socketio as well.

How does this technology accomplish what it does?

This technology uses two specific class methods to run a socketio supportive Flask server and register event listeners for socket communication. The two methods, `run` & `on`, and their functionality is described below.

  - `method on(Line#260)`

    This method is used to register a SocketIO event handler.

    This event handler inturn listens for the event to be fired by the front end and executes the custom functionality defined by the user. The very last function calls to execute, made from `on`, is `set_handler`. `set_handler` registers the event by adding the name of the event to a data structure.

    Here is a snapshot of the `set_handler` function from the source code.

```
def set_handler(handler):
    if namespace not in self.handlers:
        self.handlers[namespace] = {}
    self.handlers[namespace][event] = handler
    return handler
```

This snapshot is from the following source code:
https://github.com/miguelgrinberg/python-socketio/blob/master/socketio/server.py

- `method run(Line#518)`

  This method is responsible for establishing a TCP connection on a custom host and port if provided, if not it defaults it to `host = "127.0.0.1" & port = 5000`. This method also is the point where the program can indicate if to run it in debug mode or not. If debug mode is set to True, then the backtrace of the stack is displayed in the browser itself for debugging purposes.

  Depending on the `async_mode` provided by the user, the app is run under one of the four categories, `"threading", "eventlet", "gevent" or "gevent_uwsgi"`. If the `async_mode` is not provided, the server is tried running in a hierarchy for whichever it succeeds.

  Here is a snapshot of the documentation, listing the order of hierarchy for running the server.

```
:param async_mode: The asynchronous model to use. See the Deployment
                   section in the documentation for a description of the
                   available options. Valid async modes are "threading",
                   "eventlet", "gevent" and "gevent_uwsgi". If this
                   argument is not given, "eventlet" is tried first, then
                   "gevent_uwsgi", then "gevent", and finally "threading".
                   The first async mode that has all its dependencies
                   installed is then one that is chosen.
```

  This snapshot is from the following source code:
  https://github.com/miguelgrinberg/python-socketio/blob/master/socketio/server.py

- `join_room(Line#887)`

What does this technology accomplish for us?

`join_room` function allows us to maintain a list of users in a certain chat room and add new users to the room they want to be added.

How does this technology accomplish what it does?

The `join_room` function essentially takes three parameters, the room the user has to join, user's session id and the namespace of the room to join. The `join_room` function then calls on to the `enter_room` function from socketio.

The `enter_room` function instantiates a `base manager class` which initially creates an empty dictionary calling it rooms to store all the rooms the user has created. Once a user requests to join a room this function checks if the namespace requested by the user exists if not it creates one, then checks for the room if it doesn't exist it creates one. The function refers to each client with `eoi_sid`, if the user has not passed any `eoi_sid` (in our case we don't) it creates an `eoi_sid` and adds it to the room.

This function essentially helps us by maintaining a data structure in the memory for all the rooms and adds a user to the list when the user wants to join a room. The source code for `enter_room` can be found below.

```
def enter_room(self, sid, namespace, room, eio_sid=None):
    """Add a client to a room."""
    if eio_sid is None and namespace not in self.rooms:
        raise ValueError('sid is not connected to requested namespace')
    if namespace not in self.rooms:
        self.rooms[namespace] = {}
    if room not in self.rooms[namespace]:
        self.rooms[namespace][room] = bidict()
    if eio_sid is None:
        eio_sid = self.rooms[namespace][None][sid]
    self.rooms[namespace][room][sid] = eio_sid
```

This snapshot is from the following source code:
https://github.com/miguelgrinberg/python-socketio/blob/master/socketio/base_manager.py

- `leave_room(Line#913)`

What does this technology accomplish for us?

`leave_room` function allows us to modify a list of users in a certain chat room and delete a user from the room, when requested.

How does this technology accomplish what it does?

`leave_room` deletes the user's sid from the room if the room and namespace exists. After deleting the user from the room it checks if there are any more users present in the room if not the room and the namespace is deleted as well.

This function essentially helps us by maintaining a data structure and deleting the users from a certain room. The source code for `leave_room` can be found below.

```
def leave_room(self, sid, namespace, room):
    """Remove a client from a room."""
    try:
        del self.rooms[namespace][room][sid]
        if len(self.rooms[namespace][room]) == 0:
            del self.rooms[namespace][room]
            if len(self.rooms[namespace]) == 0:
                del self.rooms[namespace]
    except KeyError:
        pass
```

This snapshot is from the following source code:
https://github.com/miguelgrinberg/python-socketio/blob/master/socketio/base_manager.py

- `emit(Line#769)`

What does this technology accomplish for us?

Emit helps us by sending a message to multiple or single clients. We use it to let all the clients connect in real time when a user sends a message or when a user likes a specific post.

How does this technology accomplish what it does?

Firstly, the function checks if the namespace and room exist in the data structure. If not, the function does not proceed.

A function `get_participants`, which returns an iterable list of all client ids of the clients present in the room.

A loop is used to iterate over the list of participants and the `_emit_internal` function is used to send the message. The message is sent as a json packet. The packet is encoded using the .encode() function and after encoding it is sent to the specified client. The source code for `_emit_internal` can be found below.

```
def _emit_internal(self, eio_sid, event, data, namespace=None, id=None):
    """Send a message to a client."""
    # tuples are expanded to multiple arguments, everything else is sent
    # as a single argument
    if isinstance(data, tuple):
        data = list(data)
    elif data is not None:
        data = [data]
    else:
        data = []
    self._send_packet(eio_sid, packet.Packet(
        packet.EVENT, namespace=namespace, data=[event] + data, id=id))
```

This snapshot is from the following source code:
https://github.com/miguelgrinberg/python-socketio/blob/master/socketio/server.py

Licensing

Flask-SocketIO is licensed under the MIT License.

The MIT License provides permissions for the following,

- Commercial Use
- Modification
- Distribution
- Private Use

The limitations of the MIT License are,

- Liability
- Warranty

The conditions of the MIT license are to preserve copyright and license notices.

Below is the actual license included along with the source code for Flask-SocketIO.

```
The MIT License (MIT)

Copyright (c) 2014 Miguel Grinberg

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.
```