# Open-Source Report

Proof of knowing your stuff in CSE312

## Guidelines

Provided below is a template you must use to write your reports for your project.

Here are some things to note when working on your report, specifically about the **General Information & Licensing** section for each technology.

- **Code Repository**: Please link the code and not the documentation. If you'd like to refer to the documentation in the **Magic** section, you're more than welcome to, but we need to see the code you're referring to as well.
- **License Type**: Three letter acronym is fine.
- **License Description**: No need for the entire license here, just what separates it from the rest.
- **License Restrictions**: What can you *not* do as a result of using this technology in your project? Some licenses prevent you from using the project for commercial use, for example.

Also, feel free to extend the cell of any section if you feel you need more room.

If there's anything we can clarify, please don't hesitate to reach out! You can reach us using the methods outlined on the course website or see us during our office hours.

# Socket.IO - WebSockets

## General Information & Licensing

| | |
|---|---|
| Code Repository | https://github.com/socketio/socket.io |
| License Type | The MIT License |
| License Description | https://github.com/socketio/socket.io/blob/main/LICENSE<br><br>(The MIT License)<br><br>Copyright (c) 2014-2018 Automattic <dev@cloudup.com><br><br>Permission is hereby granted, free of charge, to any person obtaining<br>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,<br>distribute, sublicense, and/or sell copies of the Software, and |

| | |
|---|---|
| | |
| License Restrictions | ● No Warranty<br>● No Liability |

## Purpose

We used Socket.io library for our websocket connection. Socket.io is a library that allows for real-time, bidirectional communication between web clients and servers. It enables applications to use WebSockets, a communications protocol that provides full-duplex communication channels over a single TCP connection. This means that, with Socket.io, you can create applications that can send and receive data in real-time, without the need for constant polling to check for updates.

It is built on top of the WebSocket protocol and included safety features such as automatic reconnection. This library has implementations in many languages, and in our case, we will be using it along with JavaScript (Node.js).

Socket.io will be specifically used to start up the WebSocket server and allow messages to be sent and received from the server to the client. These messages will include important information for our auction functionality. The **.on()** and **.emit()** functions will be used to implement the needed functionality. **.on()** will listen and receive messages to all connected “emitters” and **.emit()** will send messages to all the designated “listeners”.

In our implementation, we use the **.on()** method to listen for the connect event, which is emitted whenever a new client connects to the server. In that case, the callback function might be used to log a message or update the user interface to indicate that a new client has joined.

## How functionality was used in our group project (auction.js):

- https://github.com/Prakshal-Jain/NFTY/blob/main/backend/routes/auction.js
- Server module was imported from "socket.io" in auction.js [Line 13]
- A server instance named "**io**" is created using "***new Server(server)***" from the previously imported element. [Line 14]
- **io.on('connect', (socket) => {...})** on [Line 17] listens to the 'connect' event emitted by the client when the websocket connection is established.
    - We declare **socket.on(<event name>, callback)** [Line 18] to listen to specific events. For example, we used **socket.on()** method to listen to a **new-bid** event, and then its callback function is invoked when a client places a new bid by emitting the event from the frontend. We can further process this data and save it in the database inside the callback function.
    - We also used **io.emit(event, …args)** to trigger an event on the server, which will then be broadcasted to all connected clients. For example, after we receive a **new-bid** event (as described above), we use **io.emit(`auction_list#${item_name}`, { status: 200, message: new_auc_list });** [Line 102] to send the list of all bids to the client and update their user interfaces (frontend) accordingly, in real-time.
        - Note: As mentioned in the *Project Requirements*, "Multiple auctions must be able to take place simultaneously without interfering with each other" - simply using a hard-coded event name doesn't work because it will send the auction_list to all the active users. However, we only want to send it to the users who joined the auction. So, we use JavaScript formatted string (`` `**auction_list#${item_name}**` ``) to make different event name for different auctions. This helps to seamlessly hold multiple auctions in real-time without interfering with each other.
- **io.on('disconnect', (socket) => {...})** on [Line 149] listens to the 'disconnect event emitted by the client when the websocket connection is lost/disconnected.

## Background:

The Socket.io library uses the **ws** module (see the dependencies on https://github.com/socketio/socket.io/blob/main/package-lock.json#L1408). This module is utilized by Socket.io for the GUID string, 101 response, payload length parsing, and masking.

## How the Socket.io lobrary relates to our **Homework 3** and **WebSocket handshake** implementation in the library:

- In our **homework-3**, we implemented WebSocket handshake, in which we used the **GUID string** (*258EAFA5-E914-47DA-95CA-C5AB0DC85B11*). In this module, the **GUID** is exported on Line-6 and is imported and used on Line-372. The functions in the library that **upgrades the connection to WebSocket** and does the websocket handshake are defined on Line-230 and Line-356.

- In our **homework-3**, we read the websocket frames that were sent back and forth between the client and server which consisted of the payload data as well as additional information that would be used to parse the frame. Each frame was read **byte-by-byte** to extract the **Masking-key/MASK, payload length, and payload data**.

## Understanding part by part implementation of websocket in the library

### Masking:

- When the MASK was set to 1, the Masking-key would be included in the frame which would be XOR'ed with every 4 bytes of the entire payload data to decode it.
    - In the **ws** module, the mask is compiled at the same time as the payload length starting on Line 68. When the first bit shows that no Masking is required, it will set the **Masking-Key** to "0000" on Line 83 and move the offset to start at the beginning of the payload data.
    - Once the payload length is determined, we check again if Masking is required for the frame, and if not, we return the parsed frame with a mask of "0000"
    - If the mask is required, we add those masking bytes into the parsing response on Lines 131-134 where we utilize that previous offset to determine the exact location of the Masking-Key bytes.
    - With the **payload length, offset, and Masking-Key** all determined, the code can then apply the mask to the rest of the frame to decode it in Line 139 using "**applyMask**" which inherited its properties from **"mask"** on Line 12.
        - In "**buffer-util.js**", **mask** is defined on Line 115 where it uses the function "**_mask**" which is defined on Line 41. This function loops through the separated payload data and applies the mask, cycling through each of the 4 bytes in Line 43. This XOR's the payload data

to decrypt it and then returns said data.

- In our **homework-3**, we sent a **101 switching protocol** response to the client to upgrade their connection to a websocket which included the connection type and hashed GUID key.
    - In the **ws** module, the randomly generated websocket key and GUID are appended in Line 371 where they are then converted to base64 to be included in the response code.
    - Once the SHA-1 hash is created, it can be added to the switching protocol response found on Line 375
        - This header included all parts of the upgrade response code including the **Sec-WebSocket-Accept** key which was previously compiled.

## Parsing the payload based on length:

- Defined in ws module/sender.js file
- The library initially find the data length based on the type of data. These cases are defined on Lines 89-102It then sets the payloadlength as data length, and changes the value of payload length based on the cases referenced below.
- The implementation of decoding with payload length defined in the library corresponds to the implementation in **Homework-3**. It can be represented as 7,16, or 64 bits, considering the cases where:
    - Payload length <= 125
        - We go to Line-114 to start decoding.
    - 126 =< Payload length < 65536 bytes
        - On Line 109, if variable *dataLength* is more than 125 bytes then first 7 bit length (or *payloadLengt*h variable in the code) is set to be 126 and can be represented with the following 16 bits/the next 2 bytes.
            - This explains the following Line 110, where the offset adds 2 (bytes) to account for the representation of the payload length.
    - Payload length >= 65536 bytes
        - On Line 106, if variable *dataLength* is more than 65536 bytes then first 7 bit length (or *payloadLengt*h variable in the code) and can represented with the following 64 bits/the next 8 bytes.
            - This explains the following Line 107, where the offset adds 8 (bytes) to account for the representation of the payload length.

- In Line 68 of the websockets library, the frame was initialized to be read byte-by-byte.
    - The variable for data length was initialized on Line 87 which will be used to check if we require extra bytes for the payload length.
    - Lines 89 - 102 checks whether data should be treated as a string, determine the data length and making sure that it is receiving

enough data by reading from the buffer (and getting the data length from the amount of data read from said buffer).

        - Based on the bytes of the payload length, Lines 119-125 will convert the first byte to decimal to determine the initial payload length. It will check if the values are 126 or 127 which will determine whether they require the additional 2 or 8 bytes and determine the final payload length accordingly.
            - These sections will convert the bytes to payload length using "**writeUInt16BE()**" or "**writeUIntBE()**" which take in 16 bits of any number of bytes respectively and convert them to an **int** value.
                - These functions use Big endian format to write the specified bytes to an **int** value, same as how we used **from_bytes()** in homework 3.

- The socket.io library uses the **decoder** and **encoder** for websocket frames imported from **socket.io-parser** module (Line-1). The code trace of the parser is as follows:
    - The **decodePacket()** function is defined on Line-407 in socket.io library. It decodes a packet of data that has been received from a client. The packet of data is typically in the form of a string or buffer (websocket frame), and it contains information about the event that was triggered on the client, as well as any data associated with the event. This function is using the **decodeBase64Packet()** function if the packet of data that has been received from a client is encoded using the Base64 encoding scheme..
    - The **decodeBase64Packet()** function is defined on Line-438 in socket.io library. It uses **decode$1()** function.
    - The **decode$1()** is defined on Line-371 in socket.io library.

- In Socket.io, the **io.on()** method is used to **listen for events**. This means that we can use it to specify a callback function that will be executed whenever a specified event occurs.
    - https://github.com/socketio/socket.io/blob/3b7ced7af7e0a2a66392577f94af1ee5ed190ab1/lib/typed-events.ts#L102
    - The function "on" is defined on Lines 102-107.
        - The 'on' function at line 102 is used to attach an event listener to a specific event.
        - This function takes 2 arguments:
            - **ev** → [Line 103] This specifies the event for which the listener is being attached.
            - **listener** → [Line 104] This is the callback function that will be executed when the event is triggered.
        - It is a convenient way to attach event listeners to events of specific types, while ensuring type safety and consistency.
        - Line 106 returns an event connection instance for the "listener".
            - For example, If the "connect" event is received when the websocket connection with the client is established. Similarly, if the "disconnect" event is received, it means that

the client is being removed from the connection, and we can take any appropriate action in the callback.

- **io.emit()** function is used to send (broadcast) an event to the clients. It takes two arguments:
  1. '**event: string**': This specifies the name of the event that is being emitted.
  2. '**...args: any[]**': This is a list of arguments that will be passed to the event listener callback function on the server or other clients
  - https://github.com/socketio/socket.io/blob/3b7ced7af7e0a2a66392577f94af1ee5ed190ab1/lib/typed-events.ts#L128
  - The function "emit" is defined on Lines 128-133
  - The **emit<Ev>** function defined in the given link is a generic function that allows emitting events of type "**Ev**". The "**Ev**" type is constrained to be an event name that is specified as an **emit** event, as defined by the "**EmitEvents**" interface.
  - The **emit<Ev>** function takes **two** arguments:
    - "**ev: Ev**" specifies the event that is being emitted. The argument type is '**Ev**', which means it must be an event name specified as an emit event. [Line 129]
    - "**...args: Parameters<EmitEvents[Ev]>**" is a list of arguments that are passed to the event listener callback function on the server. The argument type is "**Parameters<EmitEvents[Ev]>**" which means in the specified event, the argument type must match the expected argument type from the event listener callback function. [Line 130]
    - "**Boolean**" requires the response of the function to return a boolean value indicating whether the "event" was successfully emitted or not. If the event is emitted successfully, the "**.emit()**" function returns 'true', otherwise it returns 'false'. [Line 131]
  - The implementation of the "**emit<Ev>**" function uses the "**.emit()**" function defined earlier and passes the "event" and "args" arguments to it. This allows emitting the specified "event" with the given "args" arguments.
  - Overall, the "**emit<Ev>**" function provides a convenient way to emit events of specific types while ensuring type safety and consistency. It also allows for the passing of strongly-typed arguments to the event listener callback function.

- As mentioned in the *Project Requirements* - "You are required to use WebSockets for these features! Resorting to polling or long-polling is not allowed (Please note that the SocketIO library often resorts to long-polling. You must disable this and force it to use WebSockets if you are using this library)", when we use socket.io library, by default, it uses long-polling. To solve this issue and truly use a WebSocket connection we used **var socket = io({transports: ['websocket']})** as an optional parameter to only use **'websocket'** as the mode of connection.