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.

[insert library/framework name here]

General Information & Licensing

Code Repository	https://github.com/kudos/koa-websocket
License Type	MIT License
License Description	(The MIT License) Copyright (c) 2018 Jonathan Cremin <jonathan@crem.in> 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	• Liability • Warranty

Magic ★★ ° ˚ ☽ ˚ ⌒ ˚ ★ ≡ ★ ≀

In our project, we used websocket for real-time communication for auctions. And we used the library called `koa-websocket` to handle websocket connections in our project.

Koa-Websocket and WS

We imported the koa-websocket library, and then call it with the Koa Application object as the argument.

```
const koaWebsocket = require('koa-websocket')
const app = koaWebsocket(new koa())
```

(https://github.com/kudos/koa-websocket/blob/master/index.js)

```
module.exports = function middleware(app, wsOptions, httpsOptions) {
 const oldListen = app.listen;
 app.listen = function listen(...args) {
   debug('Attaching server...');
   if (typeof httpsOptions === 'object') {
     const httpsServer = https.createServer(httpsOptions, app.callback());
     app.server = httpsServer.listen(...args);
   } else {
     app.server = oldListen.apply(app, args);
   }
   const options = { server: app.server };
   if (wsOptions) {
     Object.keys(wsOptions).forEach((key) => {
       if (Object.prototype.hasOwnProperty.call(wsOptions, key)) {
         options[key] = wsOptions[key];
       }
     });
   }
   app.ws.listen(options);
   return app.server;
 };
 app.ws = new KoaWebSocketServer(app);
 return app;
};
```

Inside the koaWebsocket function, it overrides the `listen` method of the Koa Application object. In the new `listen` method, it calls the `oldListen` method(highlighted by green) to get a server instance as default.
Then, it puts the `app.server` it just got from the `oldListen` into the `options` object along with the wsOptions, and then pass it into `app.ws.listen` function.
And `app.ws` is an instance of `KoaWebSocketServer`.

```
function KoaWebSocketServer(app) {
 this.app = app;
 this.middleware = [];
}
```

The `KoaWebSocketServer` takes the Koa Application as the argument for its construction method, and store it inside the `KoaWebSocketServer` instance. The `KoaWebSocketServer` also provides several methods, such as `listen` to listen to a port, `onConnection` to listen to `connect` evet, `use` to load middlewares.

```
const ws = require('ws');
const WebSocketServer = ws.Server;
…
KoaWebSocketServer.prototype.listen = function listen(options) {
 this.server = new WebSocketServer(options);
 this.server.on('connection', this.onConnection.bind(this));
};
```

When the `listen` method gets called, it creates `ws.Server` instance from another library called `ws`, and bind the `onConnection` method to the `connection` event of this `ws.Server` instance.

`ws` is a Node.JS library provides websocket service,
GitHub Repo: https://github.com/websockets/ws.

The `ws` library provides the `Server` class in the `ws/lib/websocket-server.js` file

Since the `koa-websocket` provides the http-server instance from Koa Application Object, the underlaying `ws` uses this http-server instance as the internal server.

```
constructor(options, callback) {
 super();
…
 if (options.server) {
  this._server = options.server;
 }
 if (this._server) {
  const emitConnection = this.emit.bind(this, 'connection');

  this._removeListeners = addListeners(this._server, {
   listening: this.emit.bind(this, 'listening'),
   error: this.emit.bind(this, 'error'),
   upgrade: (req, socket, head) => {
     this.handleUpgrade(req, socket, head, emitConnection);
   }
  });
 }

…
}
…
function addListeners(server, map) {
 for (const event of Object.keys(map)) server.on(event, map[event]);

 return function removeListeners() {
   for (const event of Object.keys(map)) {
     server.removeListener(event, map[event]);
   }
 };
}
```

And then it uses the `addListeners` to bind its listeners to the events of the `server` object.

WebSocket HandShake

When the http server receives a `upgrade` request and fires the `upgrade` event, it will be handle by the `this.handleUpgrade` method along with the req, socket, head and emitConnection as the arguments.

```
handleUpgrade(req, socket, head, cb) {
 socket.on('error', socketOnError);

 const key = req.headers['sec-websocket-key'];
 const version = +req.headers['sec-websocket-version'];
```

The code above takes the `sec-webcoket-key` and `sec-websocket-version` from the headers and store them into variable `key` and `version`.

```
if (req.method !== 'GET') {
  const message = 'Invalid HTTP method';
  abortHandshakeOrEmitwsClientError(this, req, socket, 405, message);
  return;
}
```

Then it checks if the method is `GET`, otherwise it will throw a HTTP 405 Error.

```
if (req.headers.upgrade.toLowerCase() !== 'websocket') {
  const message = 'Invalid Upgrade header';
  abortHandshakeOrEmitwsClientError(this, req, socket, 400, message);
  return;
}
```

Then it checks if the upgrade header is `websocket`, otherwise it will throw a HTTP 400 Error.

```
if (!key || !keyRegex.test(key)) {
  const message = 'Missing or invalid Sec-WebSocket-Key header';
  abortHandshakeOrEmitwsClientError(this, req, socket, 400, message);
  return;
}
```

Then it checks the key from `sec-webcoket-key` header. If the key doesn't exists or it does not match the format, it will also return a HTTP 400 Error.

```
if (version !== 8 && version !== 13) {
  const message = 'Missing or invalid Sec-WebSocket-Version header';
  abortHandshakeOrEmitwsClientError(this, req, socket, 400, message);
```

```
  return;
}

if (!this.shouldHandle(req)) {
  abortHandshake(socket, 400);
  return;
}
```

Here it uses the `shouldHandle` method to check if the ws handles this Upgrade request.

```
const secWebSocketProtocol = req.headers['sec-websocket-protocol'];
let protocols = new Set();

if (secWebSocketProtocol !== undefined) {
  try {
    protocols = subprotocol.parse(secWebSocketProtocol);
  } catch (err) {
    const message = 'Invalid Sec-WebSocket-Protocol header';
    abortHandshakeOrEmitwsClientError(this, req, socket, 400, message);
    return;
  }
}

const secWebSocketExtensions = req.headers['sec-websocket-extensions'];
const extensions = {};
```

Then it also have code to check websocket protocol header and extension headers.

```
…

this.completeUpgrade(extensions, key, protocols, req, socket, head, cb);
}
```

Then when the request passes all the checking processes, it calls the `completeUpgrade` method to complete the websocket handshake.

```
completeUpgrade(extensions, key, protocols, req, socket, head, cb) {
  //
  // Destroy the socket if the client has already sent a FIN packet.
  //
  if (!socket.readable || !socket.writable) return socket.destroy();

  if (socket[kWebSocket]) {
    throw new Error(
      'server.handleUpgrade() was called more than once with the same ' +
        'socket, possibly due to a misconfiguration'
    );
  }

  if (this._state > RUNNING) return abortHandshake(socket, 503);
```

Here it checks if the TCP Socket still active, otherwise it will abort the handshake.

```
  const digest = createHash('sha1')
```

```
    .update(key + GUID)
    .digest('base64');
```

Here it append the fixed GUID into the key from the header of the upgrade request, and use the `createHash` function to calculate the hash of the in order to return to the client to finish the handshake process.

```
const headers = [
  'HTTP/1.1 101 Switching Protocols',
  'Upgrade: websocket',
  'Connection: Upgrade',
  `Sec-WebSocket-Accept: ${digest}`
];

const ws = new this.options.WebSocket(null);
```

Here it builds the HTTP header for finishing the upgrade request and create a `option.WebSocket` instance.

```
…

//
// Allow external modification/inspection of handshake headers.
//
this.emit('headers', headers, req);

socket.write(headers.concat('\r\n').join('\r\n'));
socket.removeListener('error', socketOnError);
```

It emits the `headers` event with the `headers` and `req` as argument to build the headers and send to the client.

```
ws.setSocket(socket, head, {
  maxPayload: this.options.maxPayload,
  skipUTF8Validation: this.options.skipUTF8Validation
});
```

Here it passes the TCP socket and head to the `ws` (the option.WebSocket, which is a websocket conenction object) instance.

```
if (this.clients) {
  this.clients.add(ws);
  ws.on('close', () => {
    this.clients.delete(ws);

    if (this._shouldEmitClose && !this.clients.size) {
      process.nextTick(emitClose, this);
```

```
        }
      });
    }
```

Then it add the `ws` to the `clients` list and listen to the `close` event of the `ws`. When the `ws`'s `close` event fires, it will know that this ws connection is close and remove itself from the `clients` list.

So far, the WebSocket connection is established, and it's abstracted into the `ws` object.

```
    cb(ws, req);
  }
}
```

And then it's passed to the callback function.

Back to Koa

Back in the construction function, it passes the `emitConnection` as the callback function to the handleUpgrade method, so when the handshake is done, the `emitConnection` will be called with the ws object and the req object.

```
const emitConnection = this.emit.bind(this, 'connection');

…
addListeners(this._server, {
   listening: this.emit.bind(this, 'listening'),
   error: this.emit.bind(this, 'error'),
   upgrade: (req, socket, head) => {
     this.handleUpgrade(req, socket, head, emitConnection);
   }
  });


KoaWebSocketServer.prototype.onConnection = function onConnection(socket,
req) {
 debug('Connection received');
 socket.on('error', (err) => {
   debug('Error occurred:', err);
 });
 const fn = co.wrap(compose(this.middleware));

 const context = this.app.createContext(req);
 context.websocket = socket;
 context.path = url.parse(req.url).pathname;

 fn(context).catch((err) => {
   debug(err);
 });
};
```

So the `connection` of the WebSocket Server object will be fired.

And the `onConnection` of the `KoaWebsocket` will be called.

```
KoaWebSocketServer.prototype.onConnection = function onConnection(socket,
req) {
 debug('Connection received');
 socket.on('error', (err) => {
   debug('Error occurred:', err);
 });
 const fn = co.wrap(compose(this.middleware));

 const context = this.app.createContext(req);
 context.websocket = socket;
 context.path = url.parse(req.url).pathname;

 fn(context).catch((err) => {
   debug(err);
 });
};
```

It's similar to how Koa handles a request from the `HTTPServer` (we wrote it in the TCP Handle report). It wraps the koa middlewares, and then use the `createContext` to create a Koa Context object from the `Request` object of Built-in HTTP Server, and put the websocket connection in the Koa Context object, and then pass to the Onion Middleware model. And it will eventually passed to the controllers in our project.

The `Option.WS` Object

During the WebSocket handshake, it creates a `Option.WS` object, which is the WebSocket Connection object.

This class is actually imported from the `websocket.js` file

```
const WebSocket = require('./websocket');
```

```
ws.setSocket(socket, head, {
 maxPayload: this.options.maxPayload,
 skipUTF8Validation: this.options.skipUTF8Validation
});
```

During the WebSocket handshake, ws.setSocket was called with the TCP Socket, head, and some options.

```
setSocket(socket, head, options) {
 const receiver = new Receiver({
   binaryType: this.binaryType,
   extensions: this._extensions,
   isServer: this._isServer,
   maxPayload: options.maxPayload,
   skipUTF8Validation: options.skipUTF8Validation
 });

 this._sender = new Sender(socket, this._extensions, options.generateMask);
```

```
  this._receiver = receiver;
  this._socket = socket;

  receiver[kWebSocket] = this;
  socket[kWebSocket] = this;

  receiver.on('conclude', receiverOnConclude);
  receiver.on('drain', receiverOnDrain);
  receiver.on('error', receiverOnError);
  receiver.on('message', receiverOnMessage);
  receiver.on('ping', receiverOnPing);
  receiver.on('pong', receiverOnPong);

  socket.setTimeout(0);
  socket.setNoDelay();

  if (head.length > 0) socket.unshift(head);

  socket.on('close', socketOnClose);
  socket.on('data', socketOnData);
  socket.on('end', socketOnEnd);
  socket.on('error', socketOnError);

  this._readyState = WebSocket.OPEN;
  this.emit('open');
}
```

The setSocket method creates a `Receiver` and `Sender` inside WebSocket object, and listen to the TCP Socket's events. In the highlighted part, it listens to the `data` event of the socket, and assigned the socketOnData as the event-listener.

```
function socketOnData(chunk) {
  if (!this[kWebSocket]._receiver.write(chunk)) {
    this.pause();
  }
}
```

In the `SocketOnData` function, it pass the chunk that it received from the TCP Socket to the `write` function of the `_receiver` of the websocket connection object.

The Receive Class is imported from the `receiver.js`

```
_write(chunk, encoding, cb) {
  if (this._opcode === 0x08 && this._state == GET_INFO) return cb();

  this._bufferedBytes += chunk.length;
  this._buffers.push(chunk);
  this.startLoop(cb);
}
```

In the `_write` function of the `Receive` Class, it adds the data chunk from the TCP socket into the `_buffers` BufferArray. And then calls the `startLoop` function.

Parse the Incoming WebSocket DataFrame

```
startLoop(cb) {
 let err;
 this._loop = true;

 do {
   switch (this._state) {
     case GET_INFO:
       err = this.getInfo();
       break;
     case GET_PAYLOAD_LENGTH_16:
       err = this.getPayloadLength16();
       break;
     case GET_PAYLOAD_LENGTH_64:
       err = this.getPayloadLength64();
       break;
     case GET_MASK:
       this.getMask();
       break;
     case GET_DATA:
       err = this.getData(cb);
       break;
     default:
       // `INFLATING`
       this._loop = false;
       return;
   }
 } while (this._loop);

 cb(err);
}
```

This is also a state machine but with less states.

```
constructor(options = {}) {
 super();

 ..

 this._state = GET_INFO;
 this._loop = false;

 ..
}
```

In the constructor function, the _state is set to `GET_INFO` by default.
So in the getLoop function, it will call the `getInfo` function.

```
getInfo() {
 if (this._bufferedBytes < 2) {
   this._loop = false;
   return;
 }
```

Here it checks if the received data less than 2 bytes, if so, it will terminate the state machine and wait for more data since the WebSocket DataFrame metadata is greater than 2 bytes.

```
const buf = this.consume(2);

if ((buf[0] & 0x30) !== 0x00) {
  this._loop = false;
  return error(
    RangeError,
    'RSV2 and RSV3 must be clear',
    true,
    1002,
    'WS_ERR_UNEXPECTED_RSV_2_3'
  );
}
```

0x30 is 00110000 in binary, it maskes the third and forth from the higher bits(which is the RSV2 and RSV3) to see if they are correctly set to 0.

```
const compressed = (buf[0] & 0x40) === 0x40;

if (compressed && !this._extensions[PerMessageDeflate.extensionName]) {
  this._loop = false;
  return error(
    RangeError,
    'RSV1 must be clear',
    true,
    1002,
    'WS_ERR_UNEXPECTED_RSV_1'
  );
}
```

0x40 is 01000000 in binary, it maskes the second bit from the higher bits(which is the RSV1) to see if they are set, this bit indicates if the message compressed.

```
this._fin = (buf[0] & 0x80) === 0x80;
this._opcode = buf[0] & 0x0f;
this._payloadLength = buf[1] & 0x7f;
```

0x80 is 10000000 in binary, it maskes the first bit from the higher bits(which is the FIN bit) to see if this is the last DataFrame of this WebSocket message.
And it uses 0x0f, which is `00001111` to mask the opcode bits.
And it maskes the the seven lowest bit of the second bytes for the payloadLength.

Then the code below process the situation that the opcode is 0000, which indiates that this is a continuation frame.

```
if (this._opcode === 0x00) {
  if (compressed) {
    this._loop = false;
    return error(
      RangeError,
      'RSV1 must be clear',
      true,
```

```
      1002,
      'WS_ERR_UNEXPECTED_RSV_1'
    );
  }

  if (!this._fragmented) {
    this._loop = false;
    return error(
      RangeError,
      'invalid opcode 0',
      true,
      1002,
      'WS_ERR_INVALID_OPCODE'
    );
  }

  this._opcode = this._fragmented;
} else if (this._opcode === 0x01 || this._opcode === 0x02) {
```

Then the code below process the situation that the opcode is 0001 and 0010, which indiates that this is a text message or binary message.

```
  if (this._fragmented) {
    this._loop = false;
    return error(
      RangeError,
      `invalid opcode ${this._opcode}`,
      true,
      1002,
      'WS_ERR_INVALID_OPCODE'
    );
  }

  this._compressed = compressed;
} else if (this._opcode > 0x07 && this._opcode < 0x0b) {
  …
} else {
  this._loop = false;
  return error(
    RangeError,
    `invalid opcode ${this._opcode}`,
    true,
    1002,
    'WS_ERR_INVALID_OPCODE'
  );
}

if (!this._fin && !this._fragmented) this._fragmented = this._opcode;
this._masked = (buf[1] & 0x80) === 0x80;
```

Here checks if the highest bit of the second byte set to 1, if so, it means the message is masked. And the bellowing code will check if the MASK bit correctly set.

```
if (this._isServer) {
  if (!this._masked) {
    this._loop = false;
    return error(
      RangeError,
```

```
        'MASK must be set',
        true,
        1002,
        'WS_ERR_EXPECTED_MASK'
      );
    }
  } else if (this._masked) {
    this._loop = false;
    return error(
      RangeError,
      'MASK must be clear',
      true,
      1002,
      'WS_ERR_UNEXPECTED_MASK'
    );
  }

  if (this._payloadLength === 126) this._state = GET_PAYLOAD_LENGTH_16;
  else if (this._payloadLength === 127) this._state = GET_PAYLOAD_LENGTH_64;
```

Here, it checks if the payloadLength equals to 126, if so, it jumps to the `GET_PAYLOAD_LENGTH_16` state to parse the `16-bits length`, if it's 127, it jumps to `GET_PAYLOAD_LENGTH_64` state to parse the `64-bits length`, otherwise it calls the `haveLength` method.

```
  else return this.haveLength();
}
```

Back to the loop of the state machine, if the state code is `GET_PAYLOAD_LENGTH_16`, it will call `getPayloadLength16`, if it's `GET_PAYLOAD_LENGTH_64`, it will call `getPayloadLength64` to get the payload length.

```
case GET_PAYLOAD_LENGTH_16:
  err = this.getPayloadLength16();
  break;
case GET_PAYLOAD_LENGTH_64:
  err = this.getPayloadLength64();
  break;

..
getPayloadLength16() {
  if (this._bufferedBytes < 2) {
    this._loop = false;
    return;
  }
```

Here it takes the following 2 bytes(16 bits) and read it as Big-Endian(Network Byte Order) unsigned int to get the payload length.

```
  this._payloadLength = this.consume(2).readUInt16BE(0);
```

```
 return this.haveLength();
}

..

getPayloadLength64() {
 if (this._bufferedBytes < 8) {
   this._loop = false;
   return;
 }

 const buf = this.consume(8);
 const num = buf.readUInt32BE(0);
```

Here it takes the following 8 bytes(64 bits) and read it as Big-Endian(Network Byte Order) unsigned big int to get the payload length.

```
 //
 // The maximum safe integer in JavaScript is 2^53 - 1. An error is
returned
 // if payload length is greater than this number.
 //
```

Here it checks if the payload length overflows

```
 if (num > Math.pow(2, 53 - 32) - 1) {
   this._loop = false;
   return error(
     RangeError,
     'Unsupported WebSocket frame: payload length > 2^53 - 1',
     false,
     1009,
     'WS_ERR_UNSUPPORTED_DATA_PAYLOAD_LENGTH'
   );
 }

 this._payloadLength = num * Math.pow(2, 32) + buf.readUInt32BE(4);
 return this.haveLength();
}
```

Both of them will jump to the `haveLength` method to parse further information

```
haveLength() {
 if (this._payloadLength && this._opcode < 0x08) {
   this._totalPayloadLength += this._payloadLength;
   if (this._totalPayloadLength > this._maxPayload && this._maxPayload > 0)
{
     this._loop = false;
     return error(
       RangeError,
       'Max payload size exceeded',
       false,
       1009,
       'WS_ERR_UNSUPPORTED_MESSAGE_LENGTH'
     );
   }
```

```
  }

  if (this._masked) this._state = GET_MASK;
  else this._state = GET_DATA;
}
```

Here it sets the state to `GET_MASK` if the MASK bit is set, otherwise, it will go the `GET_DATA` state.

```
case GET_MASK:
  this.getMask();
  break;
case GET_DATA:
  err = this.getData(cb);
  Break;
```

If the state is set to `GET_STATE`, it goes to `getMask` method.

```
getMask() {
  if (this._bufferedBytes < 4) {
    this._loop = false;
    return;
  }

  this._mask = this.consume(4);
  this._state = GET_DATA;
}
```

It reads the following 4 bytes as the MASK, then set the state to `GET_DATA`

```
getData(cb) {
  let data = EMPTY_BUFFER;

  if (this._payloadLength) {
    if (this._bufferedBytes < this._payloadLength) {
      this._loop = false;
      return;
    }

    data = this.consume(this._payloadLength);
  }
..
  if (data.length) {
    //
    // This message is not compressed so its length is the sum of the
payload
    // length of all fragments.
    //
    this._messageLength = this._totalPayloadLength;
    this._fragments.push(data);
  }

  return this.dataMessage();
}
```

The getData method will consume `_payloadLength` bytes of data from the socket, and then push the data it consumes to the `_fragments` list.

Then it calls the `dataMessage` method.

```
dataMessage() {
 if (this._fin) {
   const messageLength = this._messageLength;
   const fragments = this._fragments;

   this._totalPayloadLength = 0;
   this._messageLength = 0;
   this._fragmented = 0;
   this._fragments = [];

   if (this._opcode === 2) {
     let data;

     if (this._binaryType === 'nodebuffer') {
       data = concat(fragments, messageLength);
     } else if (this._binaryType === 'arraybuffer') {
       data = toArrayBuffer(concat(fragments, messageLength));
     } else {
       data = fragments;
     }

     this.emit('message', data, true);
   } else {
     const buf = concat(fragments, messageLength);

     if (!this._skipUTF8Validation && !isValidUTF8(buf)) {
       this._loop = false;
       return error(
         Error,
         'invalid UTF-8 sequence',
         true,
         1007,
         'WS_ERR_INVALID_UTF8'
       );
     }

     this.emit('message', buf, false);
   }
 }

 this._state = GET_INFO;
}
```

The dataMessage will concat all the data framents into a single buffer and then emit the `message` event with the data buffer, and then sets the state to `GET_INFO` for future data from the TCP scoket.

So far, the Websocket-Parsing is done.

`send` method of WebSocket

In our project, we use the `send` method of the websocket to send data through the websocket.

```
send(data, options, cb) {
 if (this.readyState === WebSocket.CONNECTING) {
   throw new Error('WebSocket is not open: readyState 0 (CONNECTING)');
 }

 if (typeof options === 'function') {
   cb = options;
   options = {};
 }

 if (typeof data === 'number') data = data.toString();

 …

 this._sender.send(data || EMPTY_BUFFER, opts, cb);
}
```

This is the `send` method in the `WebSocket` class. It checks the connection state first, and then finally pass the data into the `send` method of the _sender.

Then sender is defined in the `sender.js`

```
send(data, options, cb) {
 const perMessageDeflate =
this._extensions[PerMessageDeflate.extensionName];
 let opcode = options.binary ? 2 : 1;
 let rsv1 = options.compress;

 let byteLength;
 let readOnly;

 ..

 if (this._firstFragment) {
   this._firstFragment = false;
   if (
     rsv1 &&
     perMessageDeflate &&
     perMessageDeflate.params[
       perMessageDeflate._isServer
         ? 'server_no_context_takeover'
         : 'client_no_context_takeover'
     ]
   ) {
     rsv1 = byteLength >= perMessageDeflate._threshold;
   }
   this._compress = rsv1;
 } else {
   rsv1 = false;
   opcode = 0;
 }
```

```
 if (options.fin) this._firstFragment = true;

 ..
   this.sendFrame(
     Sender.frame(data, {
       [kByteLength]: byteLength,
       fin: options.fin,
       generateMask: this._generateMask,
       mask: options.mask,
       maskBuffer: this._maskBuffer,
       opcode,
       readOnly,
       rsv1: false
     }),
     cb
   );
}
```

It calls the frame with the data to build the WebSocket DataFrame.

```
static frame(data, options) {
 let mask;
 let merge = false;
 let offset = 2;
 let skipMasking = false;
```

Here it check the mask option

```
 if (options.mask) {
   mask = options.maskBuffer || maskBuffer;

   if (options.generateMask) {
     options.generateMask(mask);
   } else {
     randomFillSync(mask, 0, 4);
   }

   skipMasking = (mask[0] | mask[1] | mask[2] | mask[3]) === 0;
   offset = 6;
 }

 let dataLength;
```

If the input data is string, it will be transform to Buffer.

```
 if (typeof data === 'string') {
   if (
     (!options.mask || skipMasking) &&
     options[kByteLength] !== undefined
   ) {
     dataLength = options[kByteLength];
   } else {
     data = Buffer.from(data);
     dataLength = data.length;
   }
 } else {
   dataLength = data.length;
   merge = options.mask && options.readOnly && !skipMasking;
 }

 let payloadLength = dataLength;
```

Here it follows the data length rules of the WebSocket dataframe.
If the data length is greater or equal to 65536, it will use the 64bit length setting.
If the lengt is greater than 125 and less than 65536, it will use the 16bit length setting,
Otherwise, it will use the 7bit setting.

```
if (dataLength >= 65536) {
  offset += 8;
  payloadLength = 127;
} else if (dataLength > 125) {
  offset += 2;
  payloadLength = 126;
}

const target = Buffer.allocUnsafe(merge ? dataLength + offset : offset);

target[0] = options.fin ? options.opcode | 0x80 : options.opcode;
if (options.rsv1) target[0] |= 0x40;
```

Here sets the FIN bit, RSV bits and Opcode bits in the first byte.

```
target[1] = payloadLength;

if (payloadLength === 126) {
  target.writeUInt16BE(dataLength, 2);
} else if (payloadLength === 127) {
  target[2] = target[3] = 0;
  target.writeUIntBE(dataLength, 4, 6);
}
```

Here it sets the 64bit length setting and 16bit length setting.

```
if (!options.mask) return [target, data];

target[1] |= 0x80;
target[offset - 4] = mask[0];
target[offset - 3] = mask[1];
target[offset - 2] = mask[2];
target[offset - 1] = mask[3];

if (skipMasking) return [target, data];
```

Here it sets the mask, if the options.mask is set.

```
if (merge) {
  applyMask(data, mask, target, offset, dataLength);
  return [target];
}

applyMask(data, mask, data, 0, dataLength);
```

```
  return [target, data];
}
```

Then it contacts all the metadata of the Websocket DataFrame and the data together and it will be feed to the TCP socket via the `sendFrame` method.