WebSockets

Introduction

Background

The web has been largely built around the so-called request/response paradigm of HTTP. Client loads up a web page and then nothing happens until user clicks onto next page. Ajax started to make the web feel more dynamic. Still, all HTTP communication was steered by the client, which required user interaction or periodic polling to load new data from the server.

In many cases—for example, for stock prices, news reports, ticket sales, traffic patterns,medical device readings, and so on—the response could be stale by the time the browser renders the page. If you want to get the most up-to-date "real-time" information, you can constantly refresh that page manually, but that's obviously not a great solution.

With polling, the browser sends HTTP requests at regular intervals and immediately receives a response. This technique was the first attempt for the browser to deliver real-time information. However, real-time data is often unpredictable, making unnecessary requests inevitable and as a result, many connections are opened and closed needlessly in low-message-rate situations.

Moreover it has overhead of carrying unnecessary header data for each request-response and introduce latency in real time applications. It also forces server to use number of underlying TCP connections for each client : one for sending information to the client and a new one for each incoming message. All these factors clearly proves that, HTTP isn’t appropriate protocol to be used by real-time, low latency applications.

WebSocket

Web Sockets represents the next evolution of web communications a full-duplex, bidirectional communications channel that operates through a single socket over the Web. It provides a true standard that you can use to build scalable, real-time web applications. It provides full-duplex communication channels over a single TCP connection. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade Request.

Syntax of a WebSocket Request

ws-URI = "ws:" "//" host [":" port] path ["?" query]
wss-URI = "wss:" "//" host [":" port] path ["?" query]

WireShark

Wireshark is a open source network packet analyser. It captures packets in real time and display them in human-readable format. It is handy in inspecting internal structure of data packets. Wireshark captures each packet sent to or from your system.

WireShark can be used for debugging websocket requests. There are many more debugging tools to inspect websocket request-response.

WebSocket Protocol

This protocol has two parts :

- a **handshake**
- and the **data transfer**.

Handshake

In the handshake phase, details of the connection are negotiated between client and server using HTTP protocol upgrade request. It's the bridge from HTTP to WS connection. Client has to start the handshake process making an Http GET Upgrade Request to the WS Server.

Also, common headers like User-Agent, Referer, Cookie, or authentication headers might be passed as well. If any header is not understood or has an incorrect value , the server will respond with a “400 bad request” and immediately close the socket.

Handshake REQUEST format

First Line of handshake request contains **request Line**.

> **Request Line syntax** : [method,requestUri,protocolVersion]
> example : GET /webSocketDemo_war_exploded/echoServer HTTP/1.1

Extra Headers params that are sent in a web socket request :

> Sec-Websocket-version, Sec-Websocket-Key,
> Sec-Websocket-Protocol,Sec-Websocket-Extension,
> Upgrade

Brief description of each of above header property can be found at link.

Sample Request:

> GET /webSocketDemo_war_exploded/echoServer HTTP/1.1
> Host: localhost:3030
> ***Connection: Upgrade***
> Pragma: no-cache
> Cache-Control: no-cache
> ***Upgrade: websocket***
> Origin: http://localhost:3030
> ***Sec-WebSocket-Version: 13***
> User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36
> Accept-Encoding: gzip, deflate, sdch
> Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
> Cookie: JSESSIONID=63b42ee6761659a2d31e62f0673e
> ***Sec-WebSocket-Key: irRDj6MiwsHas5+9LWVLDQ==***
> ***Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits***

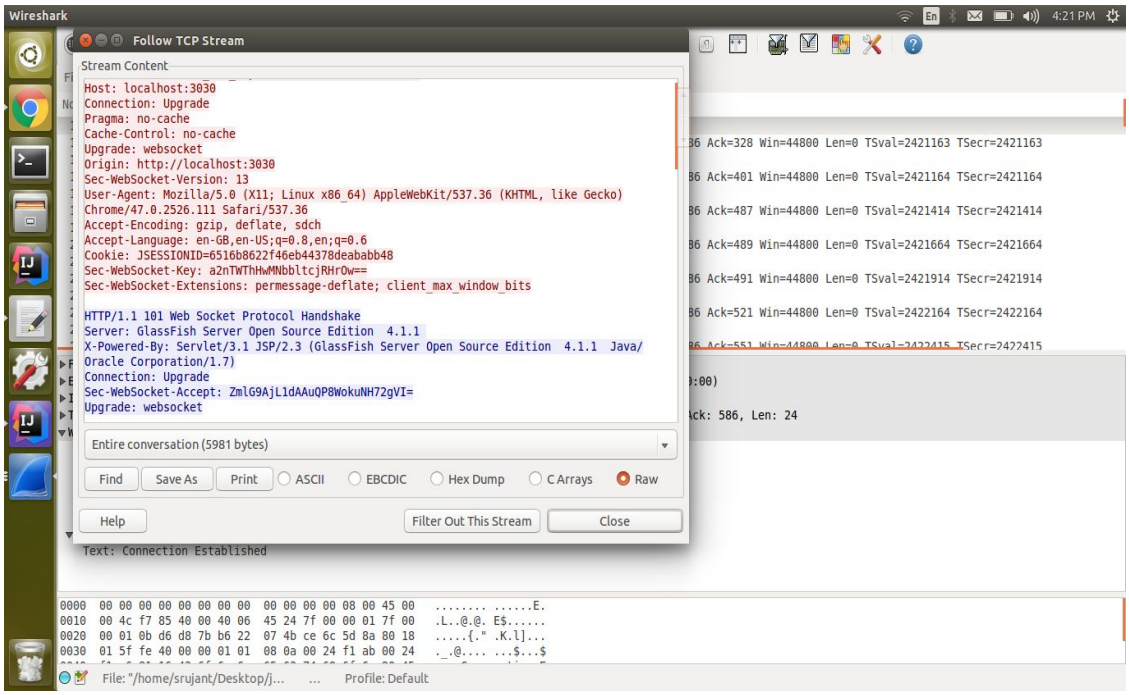
Handshake RESPONSE format

The first line of handshake response contains **status line**.

StatusLine syntax : [protocol , status Code, textual Phrase]
example : HTTP/1.1 101 Websocket Protocol Handshake

Sample Response

HTTP/1.1 ***101*** Web Socket Protocol Handshake
Server: GlassFish Server Open Source Edition 4.1.1
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.1.1 Java/Oracle Corporation/1.7)
Connection: Upgrade
Sec-WebSocket-Accept: DzDykd5cDClUOeNloO9zPNq1NNw=
Upgrade: websocket



WebSocket HandShake

Data Transfer

Once the client and server have both sent their handshakes, and if the handshake was successful, then the **data transfer** phase starts. Clients and servers transfer data back and forth in conceptual units referred to in this specification as **"messages"**. Each message is composed of one or more **frames**. Each frame belonging to the same message contains the same type of data.

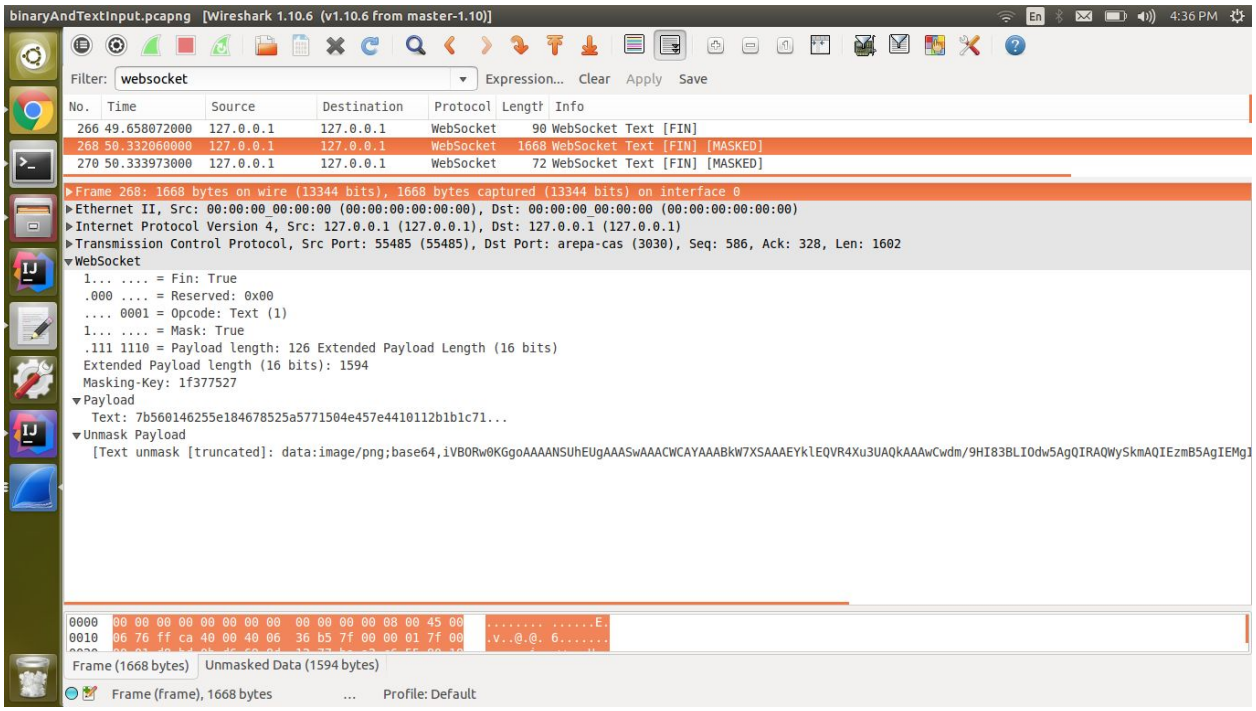
A client MUST **mask** all frames that it sends to the server. The server MUST close the connection upon receiving **a frame that is not masked.** In this case, a server may send a **Close** frame. **Masking** payload data is similar to data encryption. It protects data frame from being modified/read by any intermediate proxies.

A data frame may be transmitted by either the client or the server at any time after opening handshake completion and before that endpoint has sent a Close frame.

Each data frame received in the web socket request has certain properties[Fin flag,opcode, mask, payload length etc] associated with it. These properties are helpful in describing data frame. A higher level overview of the **frame** is clearly described at link. Each frame can contain either textual or binary data.

Types of Frames

Text Frame



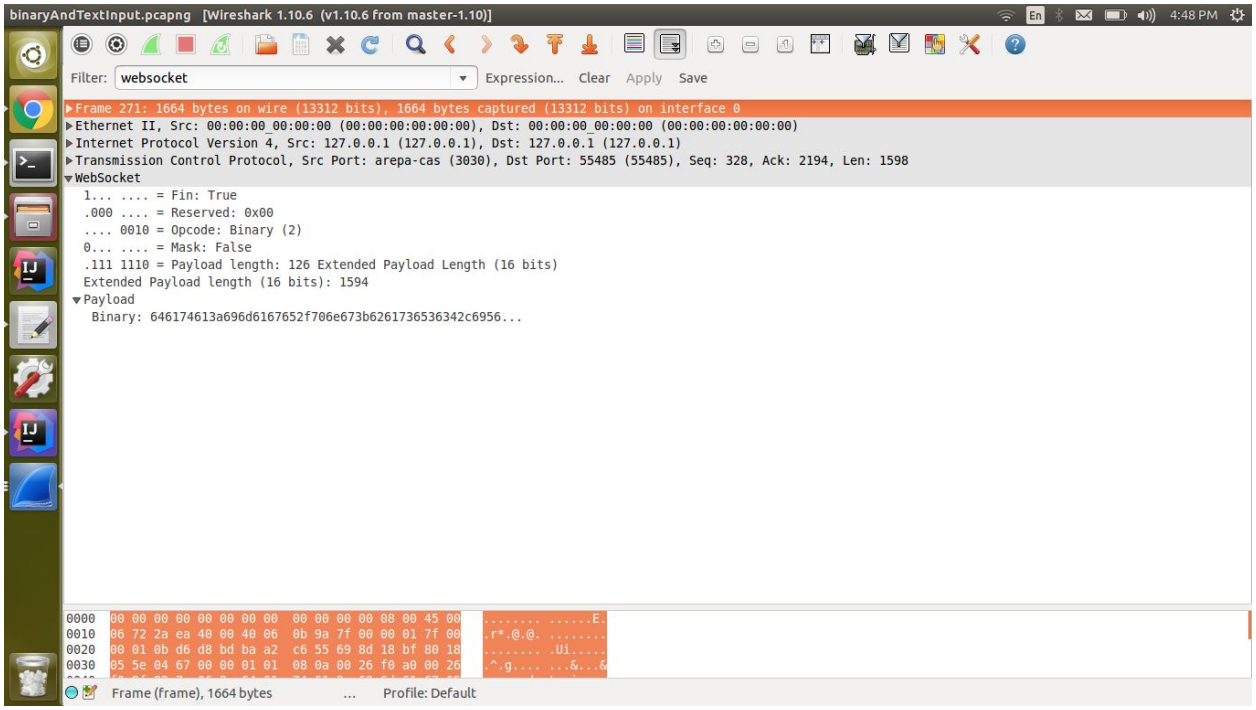
Frame carrying TextData

Above is a sample frame inspected using wireShark. Frame contains **FIN flag** set to 1, which indicates that it's an endFrame. The **opcode** value **1** indicates that the frame received is a **text Frame**. Properties **Mask** is set to true. It indicates, payload data is a encrypted data.

Payload contains masked version of payload Data. **Unmask Payload** shows actual data content inside the frame. When Mask flag is set to true, a masking key is truncated into data frame, which is used on server end for decrypting masked data.

Binary Frame

A binary frame will contain **opcode set to 2**.



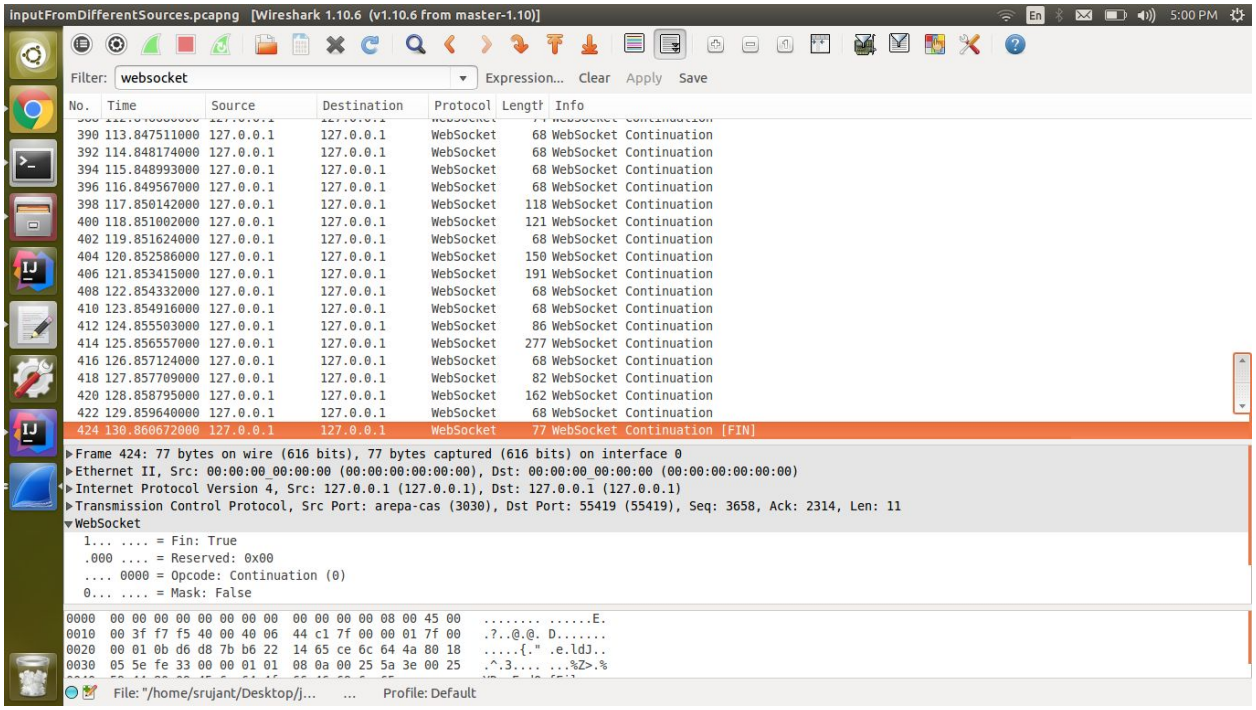
Frame with binary data

Continuation Frame

The payload data may be sent over **multiple frames**. We know the size of the message sent by the payload length of frame. Browser combines data frames together to form a single message until we receive the data frame with the **Fin flag** set to 1. A continuation frame if it exists, **will contain the 0 opcode** and FIN flag set to 1.

Continuation Frame Structure :

In the below picture, **FIN** flag of last continuation-frame is set to **1.** It indicates the **end of message**. Data get's rendered onto browser only after receiving the continuation frame with **FIN** flag set to 1.



Continuation Frame

WebSocket Client Side

Frames sent by server can be either **text Frames** or **binary Frames**. The `ws.binaryType` allows you to define in which format you'd like to obtain the binary data.
To determine type, we can use

```
e.data instanceof ArrayBuffer
e.data instanceof Blob
typeof e.data === "string"
```

By default **ws.binaryType** is set to blob. Binary data is received by client in blob format. Setting value **ws.binaryType = 'arraybuffer'** will change the format of binary Data received into arrayBuffer.

The Websocket API provided by the browser is remarkably small and simple. To initiate a new connection, we need the URL of a WebSocket resource and a few application callbacks.

```
var ws = new WebSocket('webSocket URL');
```

Basic CallBack functions implemented on client side :

onOpen

This function is invoked, when a connection is established.

onClose

Is invoked, when a connection is terminated.

onError

Is invoked, when a connection error has occurred.

onMessage

Is invoked, for each new message received from server.

WebSocket Server Side

Server endpoint classes are POJOs (Plain Old Java Objects) that are annotated with javax.websocket.server.ServerEndpoint.

Special annotations like @OnOpen, @OnClose, @OnMessage are provided to handle websocket requests.

@OnOpen

This annotation may be used on certain methods of @ServerEndpoint but only once per endpoint. It is used to decorate a method which is called once new connection is established. It may have optional Session parameter.

@OnClose

This is used only once per endpoint. It is used to decorate a method which is called once the connection is being closed. The method may have one Session parameter, one CloseReason parameter.

@OnError

This is used only once per endpoint. It is used to decorate a method which is called once Exception is being thrown by any method annotated with @OnOpen, @OnMessage and @OnClose. It may have optional Session parameter.

@OnMessage

It is used to decorate a method which is called once new message is received. Each websocket endpoint may only have one message handling method for each of the native websocket message formats: text, binary and pong.

Decoders And Encoders

WebSockets make use of Encoders/Decoders in order to transmit complex structures.

A WebSocket can use a **Decoder** to transform a text message into a Java object and then handle it in the @OnMessage method. Whenever an object is written to the Session, a Websocket will use an **Encoder** to convert the object to text and send it back to the client

Often, XML or JSON is used for the transmission of WebSocket messages, and the encoding/decoding then comes down to marshalling a Java object into XML or JSON and back.

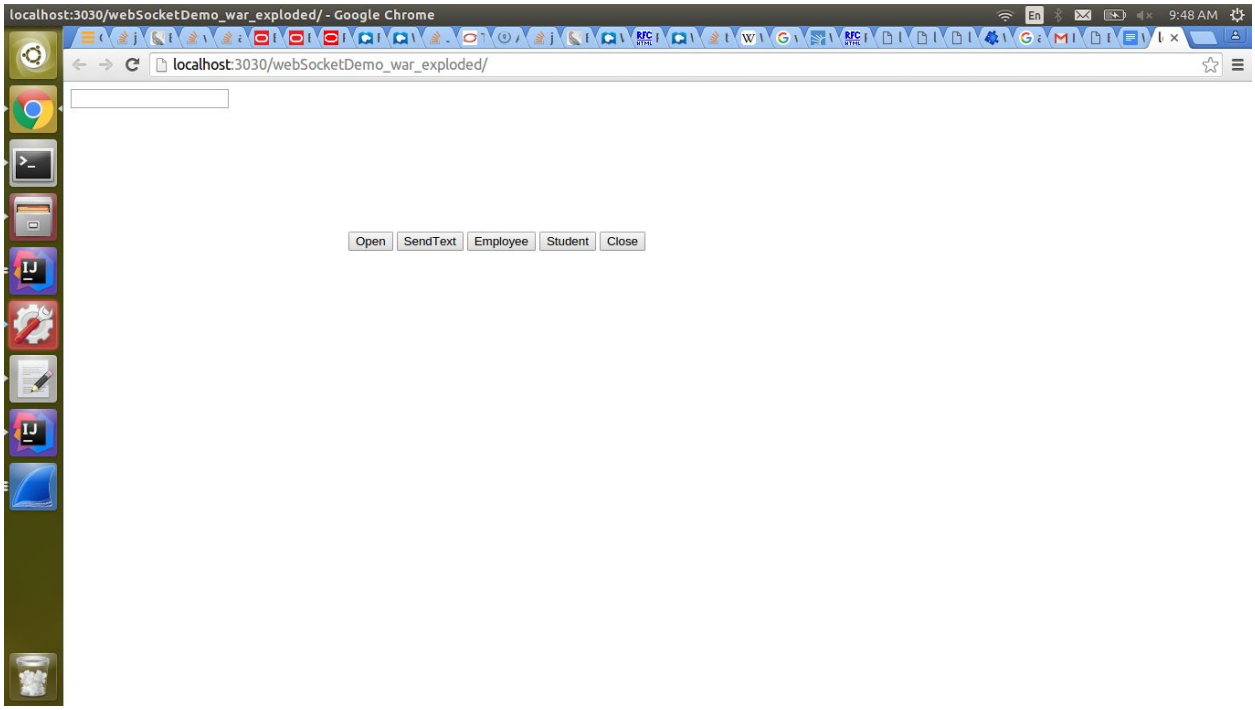
Sample Application on WebSockets :

Application Server Used : Glassfish

Application Overview

WebSocket server implemented in this sample application is similar to that of echoing server. Whatever message is sent to it, is broadcasted to clients connected to this network.

On client side index.jsp is used to initiate webSocket connection and send data to websocket server.



Index.jsp

1) **Open and Close Buttons**

They are provided to initiate and end webSocket connection.

2) **SendText**

It sends whatever data entered into textarea.

3) **Employee**

It sends a 'hello world' message as a blob type, followed by employee data as a json String to the Server.

```
webSocket.send(new Blob(["hello world"]));
var employee = {"type": "Employee", "name": "tyson", "employeeId": "1111"}
webSocket.send(JSON.stringify(employee));
```

4) **Student**

It sends student data as a json String.

```
var student = {"type": "Student", "name": "tyson", "age": 15, "collegeId": "dfsdf", "collegeName": "bvrit"};
webSocket.send(JSON.stringify(student));
```

Client

There is a standard API for webSockets in JavaScript and is supported by most modern browsers.
Client code for this application client.

Server

SeverEnd code for this application SocketServer.

SessionManager interface used in this application SessionManagerInterface and its respective implementation class SessionManager Implementation.

Frequently Asked Questions

How does WebSocket maintain persistent connection when there is no traffic flow?

Websockets make use of Ping-Pong signals to persist connection, whenever there is no traffic flow. The **Ping Frame** contains an opcode of **0x9**. The **Pong Frame** contains an opcode of **0xA.** Server sends Ping signal to end-user. Browser automatically detects it and responds with a Pong signal to server.

There is no API support on client side to send a **Ping Frame.**

How to send Request-Parameters in WebSocket?

Similar to HTTP, webSockets do have a provision to send **Path Parameters** and **Query Parameters** in the initial handshake request**.**

Path Parameters

@PathParam annotation can be used on server endpoint to retrieve path parameter values. Parameter may be of type String or any Java primitive data type. Client request URI should match URI-template.

```
@ServerEndpoint("/echoServer/{username}")
public class SocketServer{
        @OnOpen
        public void onOpen(@PathParam("username") String username, Session session )
        {
                System.out.println("Message from " + " : " + username);
        }
  }
```

Query Parameters

session.getQueryString() can be used to retrieve query parameters from request URI.

How to handle network interruptions?

If any network interruptions occur in the middle of webSocket communication, client must explicitly reinitiate WebSocket connection to the server. Data transmitted in the middle of network interruption is never recieved by the end points. In such cases the onError callback gets invoked with the corresponding message in both ends.

How to set Content-Type?

Data transmission in WebSockets can only be either **binary** or **text**. It is the business logic of the application to specify the content type in the **text** in their own format. The conversion from & to, to their own format (say json to object & vice versa) has to be explicitly performed after reading the **text** data.