Web sockets

Web sockets are defined as a two-way communication between the servers and the clients, which mean both the parties, communicate and exchange data at the same time.

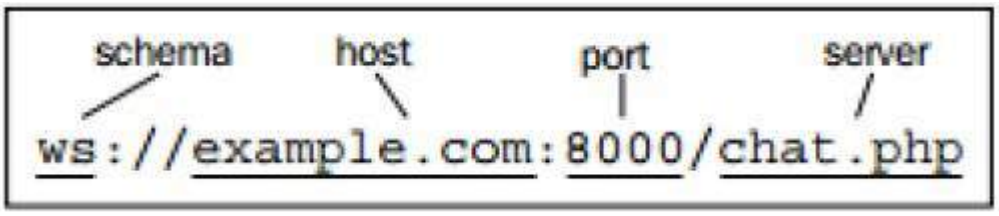
This protocol defines a full duplex communication from the ground up. Web sockets take a step forward in bringing desktop rich functionalities to the web browsers.
The main features of web sockets are as follows:

- Web socket protocol is being standardized, which means real time communication between web servers and clients is possible with the help of this protocol.
- Web sockets are transforming to cross platform standard for real time communication between a client and the server.
- This standard enables new kind of the applications. Businesses for real time web application can speed up with the help of this technology.
- The biggest advantage of Web Socket is it provides a two-way communication (fullduplex) over a single TCP connection.

URL

HTTP has its own set of schemas such as http and https. Web socket protocol also has similar schema defined in its URL pattern.

The following image shows the Web Socket URL in tokens.



Browser Support

The latest specification of Web Socket protocol is defined as **RFC 6455** – a proposed standard.

RFC 6455 is supported by various browsers like Internet Explorer, Mozilla Firefox, Google Chrome, Safari, and Opera.

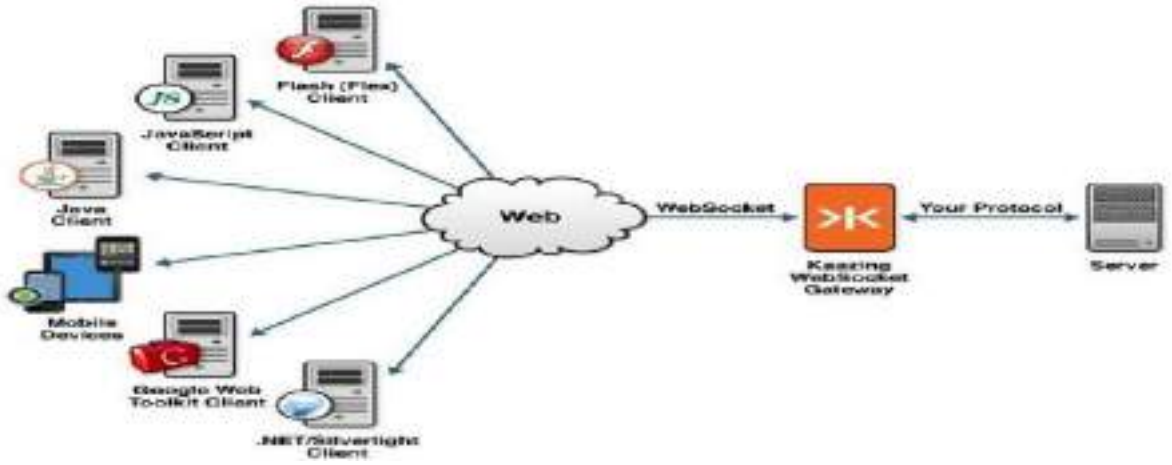
Functionalities

Web Socket represents a major upgrade in the history of web communications. Before its existence, all communication between the web clients and the servers relied only on HTTP.

Web Socket helps in dynamic flow of the connections that are persistent full duplex. Full duplex refers to the communication from both the ends with considerable fast speed.

It is termed as a game changer because of its efficiency of overcoming all the drawbacks of existing protocols.

The following diagram describes the functionalities of Web Sockets −



Web Socket connections are initiated via HTTP; HTTP servers typically interpret Web Socket handshakes as an Upgrade request.

Web Sockets can both be a complementary add-on to an existing HTTP environment and can provide the required infrastructure to add web functionality. It relies on more advanced, full duplex protocols that allow data to flow in both directions between client and server.

Events

There are four main Web Socket API **events** −

- Open
- Message
- Close
- Error

Each of the events are handled by implementing the functions like **onopen, onmessage**, **onclose** and **onerror** functions respectively. It can also be implemented with the help of addEventListener method.

The brief overview of the events and functions are described as follows −

Open

Once the connection has been established between the client and the server, the open event is fired from Web Socket instance. It is called as the initial handshake between client and server. The event, which is raised once the connection is established, is called **onopen**.

Message

Message event happens usually when the server sends some data. Messages sent by the server to the client can include plain text messages, binary data or images. Whenever the data is sent, the **onmessage** function is fired.

Close

Close event marks the end of the communication between server and the client. Closing the connection is possible with the help of **onclose** event. After marking the end of communication with the help

of **onclose** event, no messages can be further transferred between the server and the client. Closing the event can happen due to poor connectivity as well.

Error

Error marks for some mistake, which happens during the communication. It is marked with the help of **onerror** event. **Onerror** is always followed by termination of connection. The detailed description of each and every event is discussed in further chapters.

Actions

Events are usually triggered when something happens. On the other hand, actions are taken when a user wants something to happen. Actions are made by explicit calls using functions by users.

The Web Socket protocol supports two main actions, namely −

- send( )
- close( )

send ( )

This action is usually preferred for some communication with the server, which includes sending messages, which includes text files, binary data or images.

A chat message, which is sent with the help of send() action, is as follows −

```
// get text view and button for submitting the message
vartextsend=document.getElementById(“text-view”);
varsubmitMsg=document.getElementById(“tsend-button”);

//Handling the click event
submitMsg.onclick=function(){
// Send the data
socket.send(textsend.value);
}
```

Note − Sending the messages is only possible if the connection is open.

close ( )

This method stands for goodbye handshake. It terminates the connection completely and no data can be transferred until the connection is re-established.

```
vartextsend=document.getElementById(“text-view”);
varbuttonStop=document.getElementById(“stop-button”);

//Handling the click event
buttonStop.onclick=function(){
// Close the connection if open
if(socket.readyState===WebSocket.OPEN){
socket.close();
}
}
```

It is also possible to close the connection deliberately with the help of following code snippet −

```
socket.close(1000,”DeliberateConnection”);
```

Opening Connections

Once a connection has been established between the client and the server, the open event is fired from Web Socket instance. It is called as the initial handshake between client and server.

The event, which is raised once the connection is established, is called the **onopen**. Creating Web Socket connections is really simple. All you have to do is call the **WebSocket constructor** and pass in the URL of your server.

The following code is used to create a Web Socket connection −

```
// Create a new WebSocket.
var socket =newWebSocket('ws://echo.websocket.org');
```

Once the connection has been established, the open event will be fired on your Web Socket instance.

onopen refers to the initial handshake between client and the server which has lead to the first deal and the web application is ready to transmit the data.

The following code snippet describes opening the connection of Web Socket protocol −

```
socket.onopen=function(event){
console.log(“Connection established”);
// Display user friendly messages for the successful establishment of connection
var.label=document.getElementById(“status”);
label.innerHTML=”Connection established”;
}
```

It is a good practice to provide appropriate feedback to the users waiting for the Web Socket connection to be established. However, it is always noted that Web Socket connections are comparatively fast.

Handling Errors

Once a connection has been established between the client and the server, an **open** event is fired from the Web Socket instance. Error are generated for mistakes, which take place during the communication. It is marked with the help of **onerror** event. **Onerror** is always followed by termination of connection.

The **onerror** event is fired when something wrong occurs between the communications. The event **onerror** is followed by a connection termination, which is a **close** event.

A good practice is to always inform the user about the unexpected error and try to reconnect them.

```
socket.onclose=function(event){
console.log("Error occurred.");

// Inform the user about the error.
var label =document.getElementById("status-label");
```

```
label.innerHTML="Error: "+event;
}
```

When it comes to error handling, you have to consider both internal and external parameters.

- Internal parameters include errors that can be generated because of the bugs in your code, or unexpected user behavior.
- External errors have nothing to do with the application; rather, they are related to parameters, which cannot be controlled. The most important one is the network connectivity.
- Any interactive bidirectional web application requires, well, an active Internet connection.

Send & Receive Messages

The **Message** event takes place usually when the server sends some data. Messages sent by the server to the client can include plain text messages, binary data, or images. Whenever data is sent, the **onmessage** function is fired.

This event acts as a client's ear to the server. Whenever the server sends data, the **onmessage** event gets fired.

The following code snippet describes opening the connection of Web Socket protocol.

```
connection.onmessage=function(e){
varserver_message=e.data;
  console.log(server_message);
}
```

It is also necessary to take into account what kinds of data can be transferred with the help of Web Sockets. Web socket protocol supports text and binary data. In terms of Javascript, **text** refers to as a string, while binary data is represented like **ArrayBuffer**.

Web sockets support only one binary format at a time. The declaration of binary data is done explicitly as follows −

```
socket.binaryType="arrayBuffer";
socket.binaryType="blob";
```

Strings

Strings are considered to be useful, dealing with human readable formats such as XML and JSON. Whenever **onmessage** event is raised, client needs to check the data type and act accordingly.

The code snippet for determining the data type as String is mentioned below −

```
socket.onmessage=function(event){

if(typeOfevent.data===String){
console.log("Received data string");
}
}
```

JSON (JavaScript Object Notation)

It is a lightweight format for transferring human-readable data between the computers. The structure of JSON consists of key-value pairs.

Example

```
{
  name:"JamesDevilson",
  message:"HelloWorld!"
}
```

The following code shows how to handle a JSON object and extract its properties −

```
socket.onmessage=function(event){
if(typeOfevent.data===String){
//create a JSON object
varjsonObject=JSON.parse(event.data);
var username = jsonObject.name;
var message =jsonObject.message;

console.log("Received data string");
}
}
```

XML

Parsing in XML is not difficult, though the techniques differ from browser to browser. The best method is to parse using third party library like jQuery.

In both XML and JSON, the server responds as a string, which is being parsed at the client end.

ArrayBuffer

It consists of a structured binary data. The enclosed bits are given in an order so that the position can be easily tracked. ArrayBuffers are handy to store the image files.

Receiving data using ArrayBuffers is fairly simple. The operator **instanceOf** is used instead of equal operator.

The following code shows how to handle and receive an ArrayBuffer object −

```
socket.onmessage=function(event){
if(event.datainstanceofArrayBuffer){
var buffer =event.data;
console.log("Receivedarraybuffer");
}
}
```

Closing a Connection

Close event marks the end of a communication between the server and the client. Closing a connection is possible with the help of **onclose** event. After marking the end of communication with the help

of **onclose** event, no messages can be further transferred between the server and the client. Closing the event can occur due to poor connectivity as well.

The **close()** method stands for **goodbye handshake**. It terminates the connection and no data can be exchanged unless the connection opens again.

Similar to the previous example, we call the **close()** method when the user clicks on the second button.

```
vartextView=document.getElementById("text-view");
varbuttonStop=document.getElementById("stop-button");

buttonStop.onclick=function(){
// Close the connection, if open.
if(socket.readyState===WebSocket.OPEN){
socket.close();
}
}
```

It is also possible to pass the code and reason parameters we mentioned earlier as shown below.

```
socket.close(1000, "Deliberate disconnection");
```

API

API, an abbreviation of Application Program Interface, is a set of routines, protocols, and tools for building software applications.

Some important features are −

- The API specifies how software components should interact and APIs should be used when programming graphical user interface (GUI) components.
- A good API makes it easier to develop a program by providing all the building blocks.
- REST, which typically runs over HTTP is often used in mobile applications, social websites, mashup tools, and automated business processes.
- The REST style emphasizes that interactions between the clients and services is enhanced by having a limited number of operations (verbs).
- Flexibility is provided by assigning resources; their own unique Universal Resource Identifiers (URIs).
- REST avoids ambiguity because each verb has a specific meaning (GET, POST, PUT and DELETE)

Advantages of Web Socket

Web Socket solves a few issues with REST, or HTTP in general −

Bidirectional

HTTP is a unidirectional protocol where the client always initiates a request. The server processes and returns a response, and then the client consumes it. Web Socket is a bi-directional protocol where there are no predefined message patterns such as request/response. Either the client or the server can send a message to the other party.

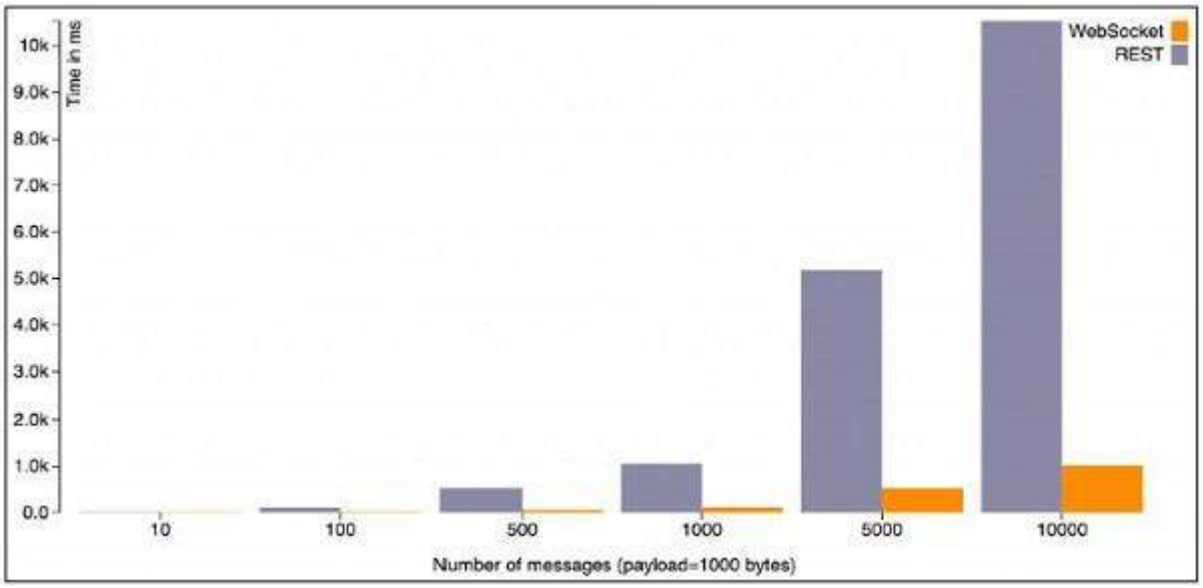
Full Duplex

HTTP allows the request message to go from the client to the server and then the server sends a response message to the client. At a given time, either the client is talking to the server or the server is talking to the client. Web Socket allows the client and the server to talk independent of each other.

Single TCP Connection

Typically, a new TCP connection is initiated for an HTTP request and terminated after the response is received. A new TCP connection needs to be established for another HTTP request/response. For Web Socket, the HTTP connection is upgraded using standard HTTP upgrade mechanism and the client and the server communicate over that same TCP connection for the lifecycle of Web Socket connection.

The graph given below shows the time (in milliseconds) taken to process N messages for a constant payload size.



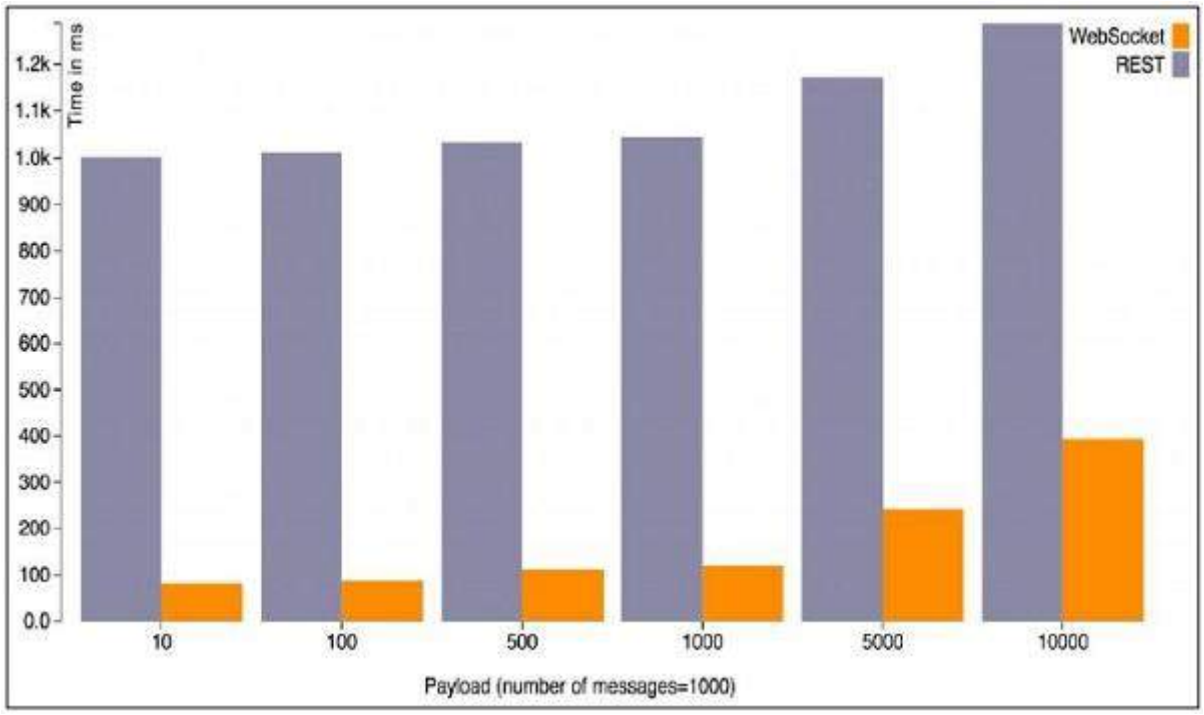
Here is the raw data that feeds this graph −

Constant payload, increasing number of messages			
Messages	**REST (in ms)**	**WebSocket (in ms)**	**x times**
10	17	13	1.31
100	112	20	5.60
500	529	68	7.78
1000	1050	115	9.13
5000	5183	522	9.93
10000	10547	1019	10.35

The graph and the table given above show that the REST overhead increases with the number of messages. This is true because that many TCP connections need to be initiated and terminated and that many HTTP headers need to be sent and received.

The last column particularly shows the multiplication factor for the amount of time to fulfil a REST request.

The second graph shows the time taken to process a fixed number of messages by varying the payload size.



Here is the raw data that feeds this graph −

Constant number of messages, increasing payload			
Payload (in bytes)	REST (in ms)	WebSocket (in ms)	x times
10	1003	81	12.38
100	1013	87	11.64
500	1032	113	9.13
1000	1044	119	8.77
5000	1173	243	4.83
10000	1289	394	3.27

This graph shows that the incremental cost of processing the request/response for a REST endpoint is minimal and most of the time is spent in connection initiation/termination and honoring HTTP semantics.

Conclusion

Web Socket is a low-level protocol. Everything, including a simple request/response design pattern, how to create/update/delete resources need, status codes etc. to be builds on top of it. All of these are well defined for HTTP.

Web Socket is a stateful protocol whereas HTTP is a stateless protocol. Web Socket connections can scale vertically on a single server whereas HTTP can scale horizontally. There

are some proprietary solutions for Web Socket horizontal scaling, but they are not based on standards. HTTP comes with a lot of other goodies such as caching, routing, and multiplexing. All of these need to be defined on top of Web Socket.

WebSockets - JavaScript Application

The following program code describes the working of a chat application using JavaScript and Web Socket protocol.

```
<!DOCTYPE html>
<htmllang="en">

<head>
<metacharset=utf-8>
<title>HTML5 Chat</title>

<body>

<sectionid="wrapper">

<header>
<h1>HTML5 Chat</h1>
</header>

<style>
#chat { width: 97%; }
.message{ font-weight: bold;}
.message:before{ content:' '; color:#bbb; font-size: 14px; }

#log {
        overflow:auto;
        max-height:300px;
        list-style: none;
        padding:0;
}

#log li {
        border-top:1px solid #ccc;
        margin:0;
        padding:10px0;
}

      body {
        font: normal 16px/20px"Helvetica Neue",Helvetica, sans-serif;
        background:rgb(237,237,236);
        margin:0;
        margin-top:40px;
        padding:0;
}
```

```
        section, header {
          display: block;
}

#wrapper {
          width:600px;
          margin:0auto;
          background:#fff;
          border-radius:10px;
          border-top:1px solid #fff;
          padding-bottom:16px;
}

        h1 {
          padding-top:10px;
}

        h2 {
          font-size:100%;
          font-style: italic;
}

        header, article >*{
          margin:20px;
}

#status {
          padding:5px;
          color:#fff;
          background:#ccc;
}

#status.fail {
          background:#c00;
}

#status.success {
          background:#0c0;
}

#status.offline {
          background:#c00;
}

#status.online {
          background:#0c0;
}

#html5badge {
          margin-left:-30px;
```

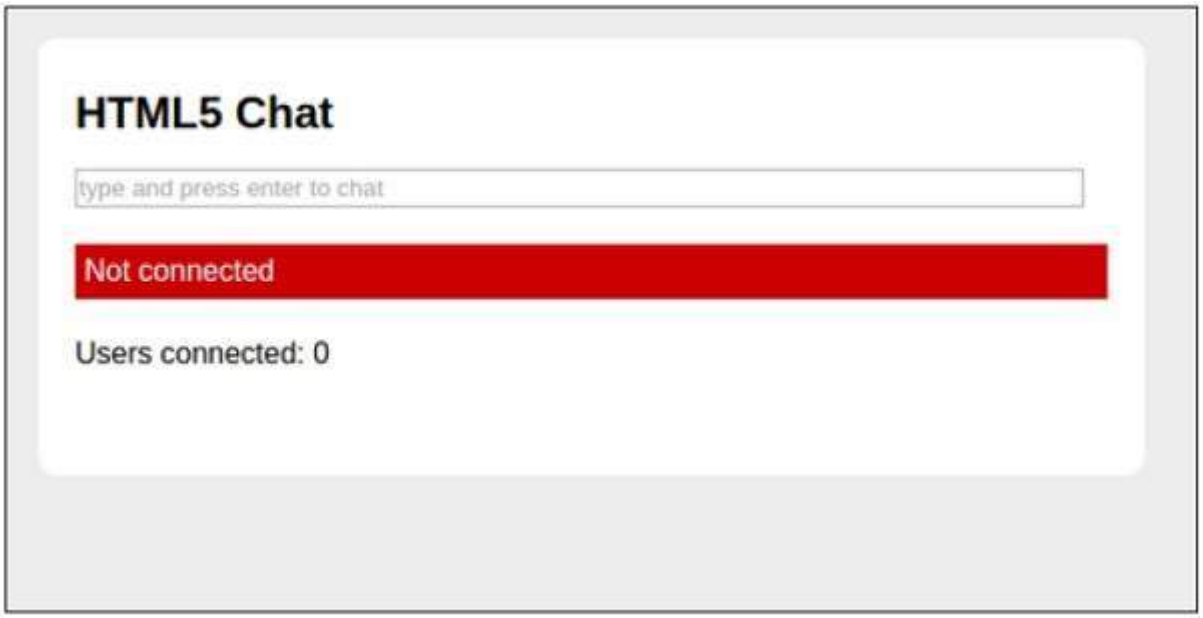
```
                border:0;
}

#html5badge img {
                border:0;
}
</style>

<article>

<formonsubmit="addMessage();returnfalse;">
<inputtype="text"id="chat"placeholder="type and press
                enter to chat"/>
</form>

<pid="status">Not connected</p>
<p>Users connected: <spanid="connected">0
</span></p>
<ulid="log"></ul>

</article>

<script>
                connected =document.getElementById("connected");
                log =document.getElementById("log");
                chat =document.getElementById("chat");
                form =chat.form;
                state =document.getElementById("status");

if(window.WebSocket===undefined){
state.innerHTML="sockets not supported";
state.className="fail";
}else{
if(typeofString.prototype.startsWith!="function"){
String.prototype.startsWith=function(str){
returnthis.indexOf(str)==0;
};
}

window.addEventListener("load",onLoad,false);
}

functiononLoad(){
varwsUri="ws://127.0.0.1:7777";
websocket=newWebSocket(wsUri);
websocket.onopen=function(evt){onOpen(evt)};
websocket.onclose=function(evt){onClose(evt)};
websocket.onmessage=function(evt){onMessage(evt)};
websocket.onerror=function(evt){onError(evt)};
}
```

```
functiononOpen(evt){
state.className="success";
state.innerHTML="Connected to server";
}

functiononClose(evt){
state.className="fail";
state.innerHTML="Not connected";
connected.innerHTML="0";
}

functiononMessage(evt){
// There are two types of messages:
// 1. a chat participant message itself
// 2. a message with a number of connected chat participants
var message =evt.data;

if(message.startsWith("log:")){
                message =message.slice("log:".length);
log.innerHTML='<li class = "message">'+
                  message +"</li>"+log.innerHTML;
}elseif(message.startsWith("connected:")){
                message =message.slice("connected:".length);
connected.innerHTML= message;
}
}

functiononError(evt){
state.className="fail";
state.innerHTML="Communication error";
}

functionaddMessage(){
var message =chat.value;
chat.value="";
websocket.send(message);
}

</script>

</section>

</body>

</head>

</html>
```

The key features and the output of the chat application are discussed below −

To test, open the two windows with Web Socket support, type a message above and press return. This would enable the feature of chat application.

If the connection is not established, the output is available as shown below.



The output of a successful chat communication is shown below.

Communicating with Server

The Web has been largely built around the request/response paradigm of HTTP. A client loads up a web page and then nothing happens until the user clicks onto the next page. Around 2005, AJAX started to make the web feel more dynamic. Still, all HTTP communication is steered by the client, which requires user interaction or periodic polling to load new data from the server.

Technologies that enable the server to send the data to a client in the very moment when it knows that new data is available have been around for quite some time. They go by names such as **"Push"** or **“Comet”**.

With **long polling**, the client opens an HTTP connection to the server, which keeps it open until sending response. Whenever the server actually has new data, it sends the response. Long polling and the other techniques work quite well. However, all of these share one problem, they carry the overhead of HTTP, which does not make them well suited for low latency applications. For example, a multiplayer shooter game in the browser or any other online game with a real-time component.

Bringing Sockets to the Web

The Web Socket specification defines an API establishing "socket" connections between a web browser and a server. In layman terms, there is a persistent connection between the client and the server and both parties can start sending data at any time.

Web socket connection can be simply opened using a constructor −

var connection = new WebSocket('ws://html5rocks.websocket.org/echo', ['soap', 'xmpp']);

ws is the new URL schema for WebSocket connections. There is also **wss**, for secure WebSocket connection the same way **https** is used for secure HTTP connections.

Attaching some event handlers immediately to the connection allows you to know when the connection is opened, received incoming messages, or there is an error.

The second argument accepts optional **subprotocols**. It can be a string or an array of strings. Each string should represent a **subprotocol** name and server accepts only one of passed **subprotocols** in the array. Accepted **subprotocol** can be determined by accessing protocol property of WebSocket object.

```
// When the connection is open, send some data to the server
connection.onopen=function(){
connection.send('Ping');// Send the message 'Ping' to the server
};

// Log errors
connection.onerror=function(error){
console.log('WebSocket Error '+ error);
};

// Log messages from the server
connection.onmessage=function(e){
console.log('Server: '+e.data);
};
```

As soon as we have a connection to the server (when the open event is fired) we can start sending data to the server using the send (your message) method on the connection object. It used to support only strings, but in the latest specification, it now can send binary messages too. To send binary data, Blob or ArrayBuffer object is used.

```
// Sending String
connection.send('your message');
```

```
// Sending canvas ImageData as ArrayBuffer
varimg=canvas_context.getImageData(0,0,400,320);
var binary =newUint8Array(img.data.length);

for(vari=0;i<img.data.length;i++){
  binary[i]=img.data[i];
}

connection.send(binary.buffer);

// Sending file as Blob
var file =document.querySelector('input[type = "file"]').files[0];
connection.send(file);
```

Equally, the server might send us messages at any time. Whenever this happens the onmessage callback fires. The callback receives an event object and the actual message is accessible via the `data` property.

WebSocket can also receive binary messages in the latest spec. Binary frames can be received in Blob or ArrayBuffer format. To specify the format of the received binary, set the binaryType property of WebSocket object to either 'blob' or 'arraybuffer'. The default format is 'blob'.

```
// Setting binaryType to accept received binary as either 'blob' or 'arraybuffer'
connection.binaryType='arraybuffer';
connection.onmessage=function(e){
  console.log(e.data.byteLength);// ArrayBuffer object if binary
};
```

Another newly added feature of WebSocket is extensions. Using extensions, it will be possible to send frames compressed, multiplexed, etc.

```
// Determining accepted extensions
console.log(connection.extensions);
```

Cross-Origin Communication

Being a modern protocol, cross-origin communication is baked right into WebSocket. WebSocket enables communication between parties on any domain. The server decides whether to make its service available to all clients or only those that reside on a set of well-defined domains.

Proxy Servers

Every new technology comes with a new set of problems. In the case of WebSocket it is the compatibility with proxy servers, which mediate HTTP connections in most company networks. The WebSocket protocol uses the HTTP upgrade system (which is normally used for HTTP/SSL) to "upgrade" an HTTP connection to a WebSocket connection. Some proxy servers do not like this and will drop the connection. Thus, even if a given client uses the WebSocket protocol, it may not be possible to establish a connection. This makes the next section even more important :)

The Server Side

Using WebSocket creates a whole new usage pattern for server side applications. While traditional server stacks such as LAMP are designed around the HTTP request/response cycle

they often do not deal well with a large number of open WebSocket connections. Keeping a large number of connections open at the same time requires an architecture that receives high concurrency at a low performance cost.

WebSockets - Security

Protocol should be designed for security reasons. WebSocket is a brand-new protocol and not all web browsers implement it correctly. For example, some of them still allow the mix of HTTP and WS, although the specification implies the opposite. In this chapter, we will discuss a few common security attacks that a user should be aware of.

Denial of Service

Denial of Service (DoS) attacks attempt to make a machine or network resource unavailable to the users that request it. Suppose someone makes an infinite number of requests to a web server with no or tiny time intervals. The server is not able to handle each connection and will either stop responding or will keep responding too slowly. This can be termed as Denial of service attack.

Denial of service is very frustrating for the end users, who could not even load a web page.

DoS attack can even apply on peer-to-peer communications, forcing the clients of a P2P network to concurrently connect to the victim web server.

Man-in-the-middle

Suppose a person **A** is chatting with his friend **B** via an IM client. Some third person wants to view the messages you exchange. So, he makes an independent connections with both the persons. He also sends messages to person **A** and his friend **B**, as an invisible intermediate to your communication. This is known as a man-in-the-middle attack.

The man-in-the-middle kind of attack is easier for unencrypted connections, as the intruder can read the packages directly. When the connection is encrypted, the information has to be decrypted by the attacker, which might be way too difficult.

From a technical aspect, the attacker intercepts a public-key message exchange and sends the message while replacing the requested key with his own. Obviously, a solid strategy to make the attacker's job difficult is to use SSH with WebSockets.

Mostly when exchanging critical data, prefer the WSS secure connection instead of the unencrypted WS.

XSS

Cross-site scripting (XSS) is a vulnerability that enables attackers to inject client-side scripts into web pages or applications. An attacker can send HTML or Javascript code using your application hubs and let this code be executed on the clients' machines.

WebSocket Native Defense Mechanisms

By default, the WebSocket protocol is designed to be secure. In the real world, the user might encounter various issues that might occur due to poor browser implementation. As time goes by, browser vendors fix any issues immediately.

An extra layer of security is added when secure WebSocket connection over SSH (or TLS) is used.

In the WebSocket world, the main concern is about the performance of a secure connection. Although there is still an extra TLS layer on top, the protocol itself contains optimizations for this kind of use, furthermore, WSS works more sleekly through proxies.

Client-to-Server masking

Every message transmitted between a WebSocket server and a WebSocket client contains a specific key, named masking key, which allows any WebSocket-compliant intermediaries to unmask and inspect the message. If the intermediary is not WebSocket-compliant, then the message cannot be affected. The browser that implements the WebSocket protocol handles masking.

Security Toolbox

Finally, useful tools can be presented to investigate the flow of information between your WebSocket clients and server, analyze the exchanged data, and identify possible risks.

Browser Developer Tools

Chrome, Firefox, and Opera are great browsers in terms of developer support. Their built-in tools help us determine almost any aspect of client-side interactions and resources. It plays a great role for security purposes.