Web Sockets

tutorialspoint
SIMPLY EASY LEARNING

www.tutorialspoint.com

About the Tutorial

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. It represents an evolution, which was awaited for a long time in client/server web technology.

Audience

This tutorial has been prepared for anyone who has a basic knowledge of Protocols and understanding of HTTP. After completing this tutorial, you will find yourself at a moderate level of expertise in understanding what makes Web Sockets different from the traditional HTTP request/response pattern.

Prerequisites

Before you start proceeding with this tutorial, we are assuming that you are already aware about the basics of JavaScript and understanding of the HTTP protocol. If you are not well aware of these concepts, then we will suggest you to go through our short tutorials on JavaScript and HTTP.

Copyright & Disclaimer

Table of Contents

Unit 1: Web Sockets – Introduction

In literal terms, handshaking can be defined as gripping and shaking of right hands by two individuals, as to symbolize greeting, congratulations, agreement or farewell. In computer science, handshaking is a process that ensures the server is in sync with its clients. Handshaking is the basic concept of Web Socket protocol.

The following diagram shows the server handshake with various clients:
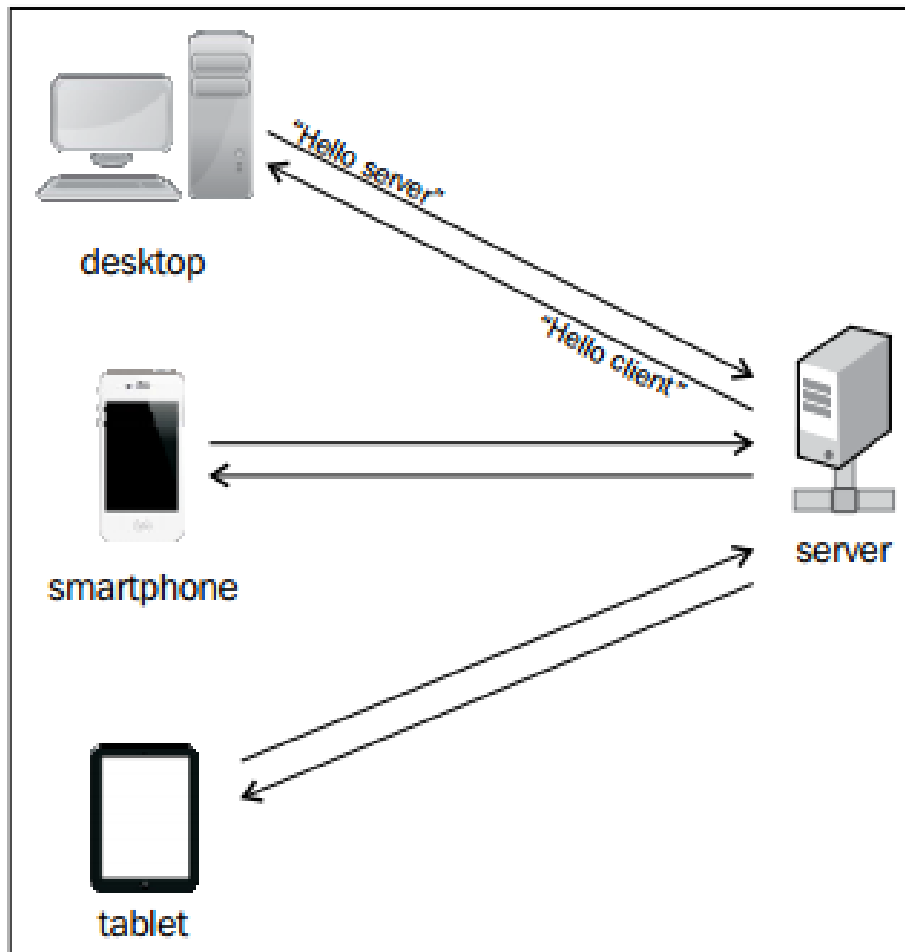
Web Sockets – Definition

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.

The key points of Web Sockets are **true concurrency** and **optimization of performance**, resulting in more responsive and rich web applications.

Description of Web Socket Protocol

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. It represents an evolution, which was awaited for a long time in client/server web technology.

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 (full duplex) 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.
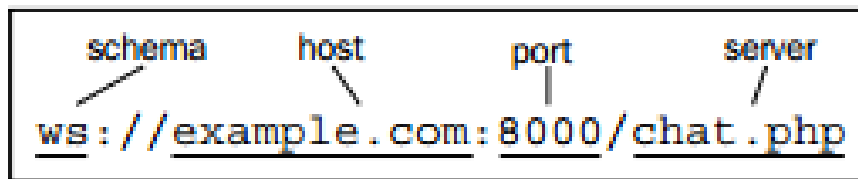
Browser Support

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.

2. Web Sockets – Duplex Communication

Before diving to the need of Web sockets, it is necessary to have a look at the existing techniques, which are used for **duplex communication** between the server and the client. They are as follows:

1. Polling
2. Long Polling
3. Streaming
4. Postback and AJAX
5. HTML5

Polling

Polling can be defined as a method, which performs periodic requests regardless of the data that exists in the transmission. The periodic requests are sent in a synchronous way. The client makes a periodic request in a specified time interval to the client. The response of the server includes available data or some warning message in it.

Long Polling

Long polling, as the name suggests, includes similar technique like polling. The client and the server keep the connection active until some data is fetched or timeout occurs. If the connection is lost due to some reasons, the client can start over and perform sequential request.
Long polling is nothing but performance improvement over polling process, but constant requests may slow down the process.

Streaming

It is considered as the best option for real-time data transmission. The server keeps the connection open and active with the client until and unless the required data is being fetched. In this case, the connection is said to be open indefinitely. Streaming includes HTTP headers which increases the file size, increasing delay. This can be considered as a major drawback.

AJAX

AJAX is based on Javascript's **XmlHttpRequest** Object. It is an abbreviated form of Asynchronous Javascript and XML. **XmlHttpRequest** Object allows execution of the Javascript without reloading the complete web page. AJAX sends and receives only a portion of the web page.

The code snippet of AJAX call with **XmlHttpRequest** Object is as follows:

```
var xhttp;
if (window.XMLHttpRequest) {
  xhttp = new XMLHttpRequest();
  } else {
```

```
// code for IE6, IE5
    xhttp =new ActiveXObject("Microsoft.XMLHTTP");
}
```

The major drawbacks of **AJAX** in comparison with **Web Sockets** are:

- They send HTTP headers, which makes total size larger.

- The communication is half-duplex.

- The web server consumes more resources.

HTML5

HTML5 is a robust framework for developing and designing web applications. The main pillars include **Mark-up, CSS3 and Javascript** APIs together.

The following diagram shows HTML5 components:

The code snippet given below describes the declaration of HTML5 and its doctype.

```
<!DOCTYPE html>
```

Why Do We Need Web Sockets?

Internet was conceived to be a collection of Hypertext Mark-up Language (HTML) pages linking one another to form a conceptual web of information. During the course of time, static resources increased in number and richer items, such as images and began to be a part of the web fabric.

Server technologies advanced which allowed dynamic server pages - pages whose content was generated based on a query.

Soon, the requirement to have more dynamic web pages lead to the availability of Dynamic Hypertext Mark-up Language (DHTML). All thanks to JavaScript. Over the following years, we saw **cross frame communication** in an attempt to avoid page reloads followed by **HTTP Polling** within frames.

However, none of these solutions offered a truly standardized cross browser solution to real-time bi-directional communication between a server and a client.

This gave rise to the need of Web Sockets Protocol. It gave rise to full-duplex communication bringing desktop-rich functionality to all web browsers.

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.

Web Socket for Developers and Architects

Importance of Web Socket for developers and architects:

- Web Socket is an independent TCP-based protocol, but it is designed to support any other protocol that would traditionally run only on top of a pure TCP connection.

- Web Socket is a transport layer on top of which any other protocol can run. The Web Socket API supports the ability to define sub-protocols: protocol libraries that can interpret specific protocols.

- Examples of such protocols include XMPP, STOMP, and AMQP. The developers no longer have to think in terms of the HTTP request-response paradigm.

- The only requirement on the browser-side is to run a JavaScript library that can interpret the Web Socket handshake, establish and maintain a Web Socket connection.

- On the server side, the industry standard is to use existing protocol libraries that run on top of TCP and leverage a Web Socket Gateway.

The following diagram describes the functionalities of Web Sockets:
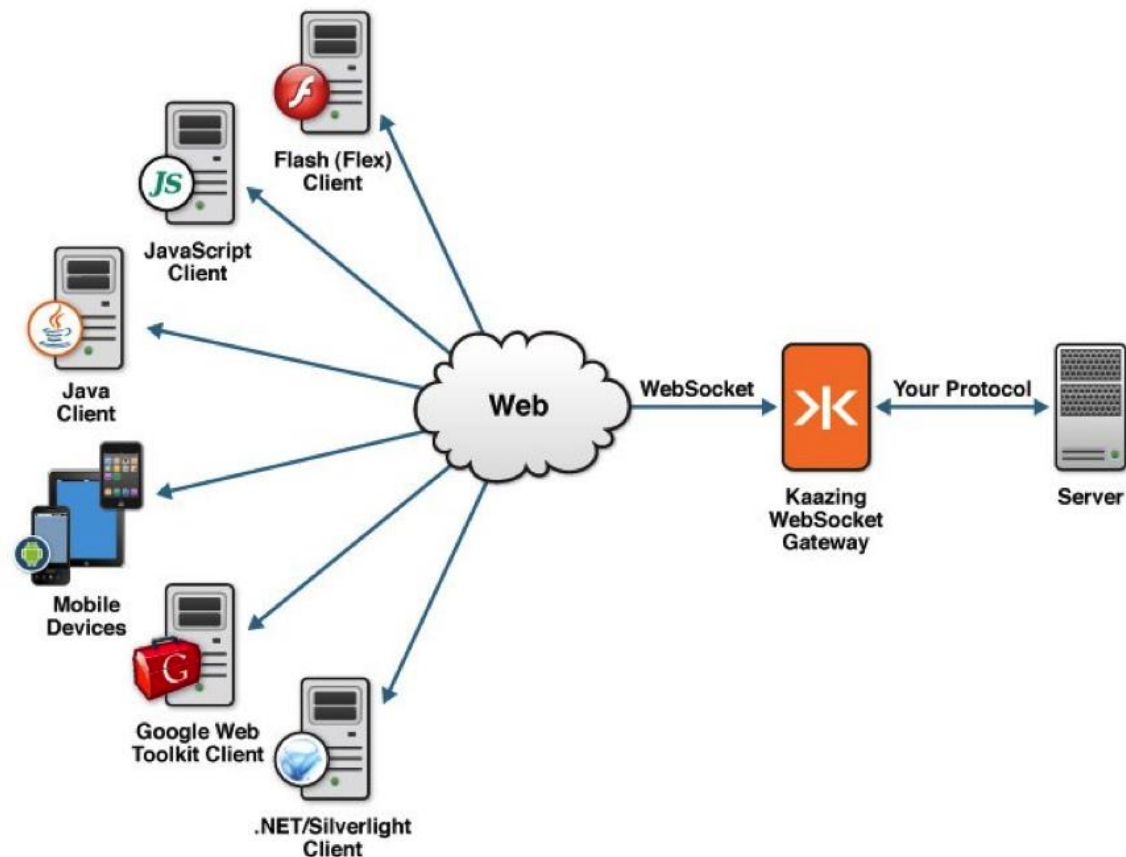
Web Socket connections are initiated via HTTP; HTTP servers typically interpret Web Socket handshakes as an Upgrade request.

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.

Functions of Web Sockets

Web Sockets provide a connection between the web server and a client such that both the parties can start sending the data.

The steps for establishing the connection of Web Socket are as follows:

1. The client establishes a connection through a process known as Web Socket handshake.

2. The process begins with the client sending a regular HTTP request to the server.

3. An Upgrade header is requested. In this request, it informs the server that request is for Web Socket connection.

4. Web Socket URLs use the **ws** scheme. They are also used for secure Web Socket connections, which are the equivalent to HTTPs.

A simple example of initial request headers is as follows:

```
GET ws://websocket.example.com/ HTTP/1.1

Origin: http://example.com

Connection: Upgrade

Host: websocket.example.com

Upgrade: websocket
```

4.　Web Sockets – Implementation

Web Sockets occupy a key role not only in the web but also in the mobile industry. The importance of Web Sockets is given below.

- Web Sockets as the name indicates, are related to the web. Web consists of a bunch of techniques for some browsers; it is a broad communication platform for vast number of devices, including desktop computers, laptops, tablets and smart phones.

- HTML5 app that utilizes Web Sockets will work on any HTML5 enabled web browser.

- Web socket is supported in the mainstream operating systems. All key players in the mobile industry provide Web Socket APIs in own native apps.

- Web sockets are said to be a full duplex communication. The approach of Web Sockets works well for certain categories of web application such as chat room, where the updates from client as well as server are shared simultaneously.
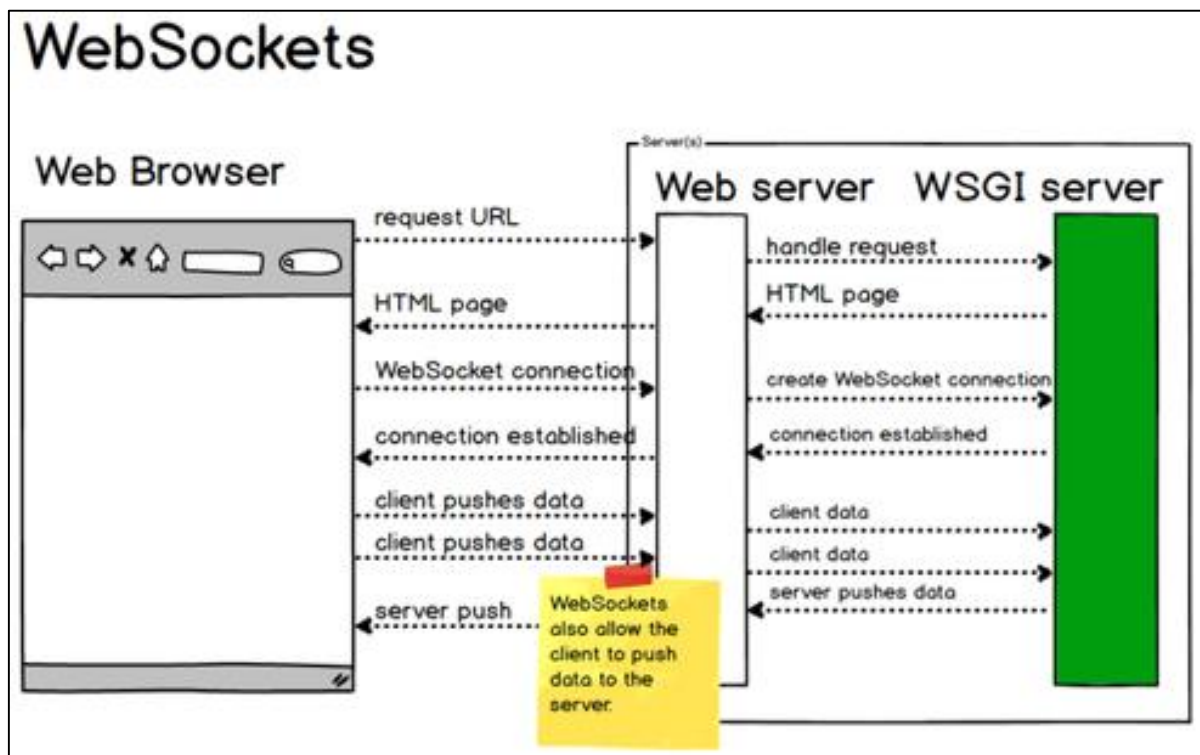
Web Sockets, a part of the HTML5 specification, allow full duplex communication between web pages and a remote host. The protocol is designed to achieve the following benefits, which can be considered as the key points:

- **Reduce unnecessary network traffic and latency** using full duplex through a single connection (instead of two).

- **Streaming through proxies and firewalls**, with the support of upstream and downstream communication simultaneously.

9

Unit 2: Web Sockets – Roles

5. Web Sockets – Events and Actions

It is necessary to initialize the connection to the server from client for communication between them. For initializing the connection, creation of Javascript object with the URL with the remote or local server is required.

```
var socket = new WebSocket(" ws://echo.websocket.org ");
```

The URL mentioned above is a public address that can be used for testing and experiments. The websocket.org server is always up and when it receives the message and sends it back to the client.

This is the most important step to ensure that application works correctly.

Web Sockets – 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.

Web Sockets – 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

var textsend = document.getElementById("text-view");

var submitMsg= 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.

```
var textsend  = document.getElementById("text-view");

var buttonStop= 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,"Deliberate Connection");
```

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 = new WebSocket('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.

The demo of the Web Socket connection established is documented in the given URL: https://www.websocket.org/echo.html

A snapshot of the connection establishment and response to the user is shown below:

Establishing an open state allows full duplex communication and transfer of messages until the connection is terminated.

Example

Building up the client-HTML5 file.

```
<!DOCTYPE html>
 <meta charset="utf-8" />
  <title>WebSocket Test</title>
   <script language="javascript" type="text/javascript">
     var wsUri = "ws://echo.websocket.org/";
      var output;
       function init() {
        output = document.getElementById("output");
         testWebSocket();
             }
       function testWebSocket() {
        websocket = new WebSocket(wsUri);
         websocket.onopen = function(evt) {
          onOpen(evt)
            };
        }
                    function onOpen(evt) {
                        writeToScreen("CONNECTED");
                        }
                      window.addEventListener("load", init, false);
```

15

```
</script>
<h2>WebSocket Test</h2>
 <div id="output"></div>
 </html>
```

The output will be as follows:

WebSocket Test

CONNECTED

The above HTML5 and JavaScript file shows the implementation of two events of Web Socket, namely:

- **onLoad**: which helps in creation of JavaScript object and initialization of connection.

- **onOpen**: establishes connection with the server and also sends the status.

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.

Checking Network Availability

Imagine that your users are enjoying your web app, when suddenly the network connection becomes unresponsive in the middle of their task. In modern native desktop and mobile applications, it is a common task to check for network availability.

The most common way of doing so is simply making an HTTP request to a website that is supposed to be up (for example, http://www.google.com). If the request succeeds, the desktop or mobile device knows there is active connectivity. Similarly, HTML has **XMLHttpRequest** for determining network availability.

HTML5, though, made it even easier and introduced a way to check whether the browser can accept web responses. This is achieved via the navigator object:

17

```
if (navigator.onLine) {

alert("You are Online");

}

else {

alert("You are Offline");

}
```

Offline mode means that either the device is not connected or the user has selected the offline mode from browser toolbar.

Here is how to inform the user that the network is not available and try to reconnect when a WebSocket close event occurs:

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

// Connection closed.

// Firstly, check the reason.

if (event.code != 1000) {

      // Error code 1000 means that the connection was closed normally.

      // Try to reconnect.

      if (!navigator.onLine) {

         alert("You are offline. Please connect to the Internet and try again.");

         }

      }

}
```

Demo for receiving error messages

The following program explains how to show error messages using Web Sockets:

```
<!DOCTYPE html>

 <meta charset="utf-8" />

  <title>WebSocket Test</title>

   <script language="javascript" type="text/javascript">

     var wsUri = "ws://echo.websocket.org/";

      var output;

       function init() {

        output = document.getElementById("output");

         testWebSocket();

             }

       function testWebSocket() {

        websocket = new WebSocket(wsUri);
```

```
      websocket.onopen = function(evt) {
  onOpen(evt)
   };
    websocket.onclose = function(evt) {
     onClose(evt)
    };
          websocket.onerror = function(evt) {
           onError(evt)
             };
          }
            function onOpen(evt) {
               writeToScreen("CONNECTED");
                doSend("WebSocket rocks");
               }
                function onClose(evt) {
                 writeToScreen("DISCONNECTED");
               }


                function onError(evt) {
                    writeToScreen('<span style="color:
red;">ERROR:</span> ' + evt.data);
                 }  function doSend(message) {
                  writeToScreen("SENT: " + message);
websocket.send(message);
               }
               function writeToScreen(message) {
                var pre = document.createElement("p");
pre.style.wordWrap = "break-word"; pre.innerHTML = message;
output.appendChild(pre);
                }
                 window.addEventListener("load", init, false);
                  </script>
                  <h2>WebSocket Test</h2>
                   <div id="output"></div>
                   </html>
```

The output is as follows:

WebSocket Test

ERROR: undefined

DISCONNECTED

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){

    var server_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(typeOf event.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: "James Devilson",
message: "Hello World!"
}
```

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

```
socket.onmessage= function(event)
{
if(typeOf event.data === String )
   {
       //create a JSON object
     var jsonObject= 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:

```javascript
socket.onmessage= function(event)
{
if(event.data instanceof ArrayBuffer )
    {
        var buffer = event.data;
       console.log("Received arraybuffer");
    }
}
```

Demo Application

The following program code shows how to send and receive messages using Web Sockets.

```html
<!DOCTYPE html>
 <meta charset="utf-8" />
  <title>WebSocket Test</title>
   <script language="javascript" type="text/javascript">
     var wsUri = "ws://echo.websocket.org/";
      var output;
       function init() {
        output = document.getElementById("output");
         testWebSocket();
            }
      function testWebSocket() {
       websocket = new WebSocket(wsUri);
        websocket.onopen = function(evt) {
         onOpen(evt)
          };
            websocket.onmessage = function(evt) {
                   onMessage(evt)
             };
                  websocket.onerror = function(evt) {
                   onError(evt)
                    };
                  }
                     function onOpen(evt) {
                        writeToScreen("CONNECTED");
```

```
                    doSend("WebSocket rocks");

                }
          function onMessage(evt) {

                writeToScreen('<span style="color: blue;">RESPONSE: '
+ evt.data+'</span>'); websocket.close();

                }

                function onError(evt) {

                    writeToScreen('<span style="color:
red;">ERROR:</span> ' + evt.data);

                }  function doSend(message) {

                 writeToScreen("SENT: " + message);
websocket.send(message);

                }

                function writeToScreen(message) {

                 var pre = document.createElement("p");
pre.style.wordWrap = "break-word"; pre.innerHTML = message;
output.appendChild(pre);

                }

                 window.addEventListener("load", init, false);

                 </script>

                 <h2>WebSocket Test</h2>

                  <div id="output"></div>

                  </html>
```

The output is shown below.

WebSocket Test

CONNECTED

SENT: WebSocket rocks

RESPONSE: WebSocket rocks

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.

```javascript
var textView = document.getElementById("text-view");

var buttonStop = 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.

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

The following code gives a complete overview of how to close or disconnect a Web Socket connection:

```html
<!DOCTYPE html>

 <meta charset="utf-8" />

  <title>WebSocket Test</title>

   <script language="javascript" type="text/javascript">

     var wsUri = "ws://echo.websocket.org/";

      var output;

       function init() {

        output = document.getElementById("output");

         testWebSocket();

           }

        function testWebSocket() {
```

```
    websocket = new WebSocket(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)
                };
              }
                 function onOpen(evt) {
                     writeToScreen("CONNECTED");
                      doSend("WebSocket rocks");
                     }
                      function onClose(evt) {
                       writeToScreen("DISCONNECTED");
                     }
                  function onMessage(evt) {
                       writeToScreen('<span style="color: blue;">RESPONSE: '
+ evt.data+'</span>'); websocket.close();
                        }
                        function onError(evt) {
                           writeToScreen('<span style="color:
red;">ERROR:</span> ' + evt.data);
                        }   function doSend(message) {
                         writeToScreen("SENT: " + message);
websocket.send(message);
                     }
                     function writeToScreen(message) {
                      var pre = document.createElement("p");
pre.style.wordWrap = "break-word"; pre.innerHTML = message;
output.appendChild(pre);
                      }
                       window.addEventListener("load", init, false);
```

```
</script>

<h2>WebSocket Test</h2>

 <div id="output"></div>

 </html>
```

The output is as follows:

WebSocket Test
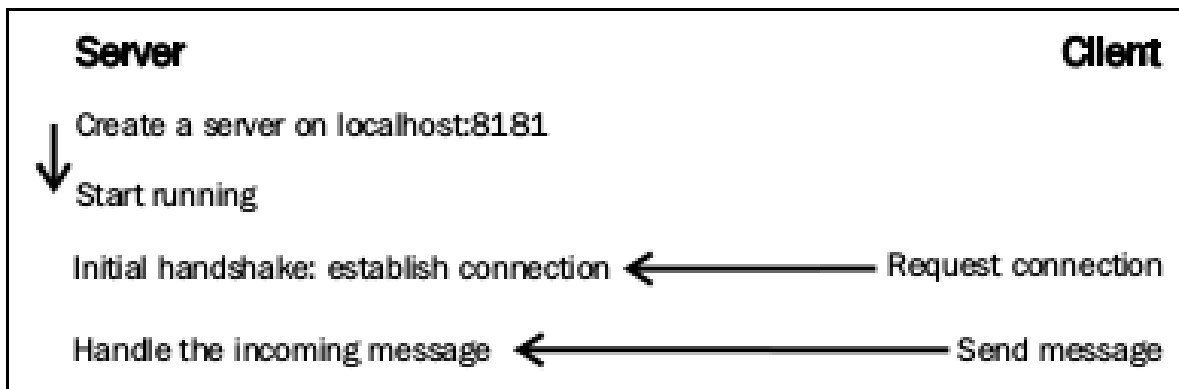
CONNECTED

SENT: WebSocket rocks

RESPONSE: WebSocket rocks

DISCONNECTED

Unit 3: JavaScript & Web Sockets API

A Web Socket server is a simple program, which has the ability to handle Web Socket events and actions. It usually exposes similar methods to the Web Socket client API and most programming languages provide an implementation. The following diagram illustrates the communication process between a Web Socket server and a Web Socket client, emphasizing the triggered events and actions.

The following diagram shows a Web Socket server and client event triggering:

Connecting to the Web Server

The Web Socket server works in a similar way to the Web Socket clients. It responds to events and performs actions when necessary. Regardless of the programming language used, every Web Socket server performs some specific actions.

It is initialized to a Web Socket address. It handles **OnOpen**, **OnClose**, and **OnMessage** events, and sends messages to the clients too.

Creating a Web Socket Server Instance

Every Web Socket server needs a valid host and port. An example of creating a Web Socket instance in server is as follows:

```
var server = new WebSocketServer("ws://localhost:8181");
```

Any valid URL can be used with the specification of a port, which was not used earlier. It is very useful to keep a record of the connected clients, as it provides details with different data or send different messages to each one.

Fleck represents the incoming connections (clients) with the **IwebSocketConnection** interface. Whenever someone connects or disconnects from our service, empty list can be created or updated.

```
var clients = new List<IWebSocketConnection>();
```

After that, we can call the **Start** method and wait for the clients to connect. After starting, the server is able to accept incoming connections. In Fleck, the Start method needs a parameter, which indicates the socket that raised the events:

```
server.Start(socket) =>
{
});
```

OnOpen Event

The **OnOpen** event determines that a new client has requested access and performs an initial handshake. The client should be added to the list and probably the information should be stored related to it, such as the IP address. Fleck provides us with such information, as well as a unique identifier for the connection.

```
server.Start(socket) =>
{
    socket.OnOpen = () =>
    {
        // Add the incoming connection to our list.
        clients.Add(socket);
    }
// Handle the other events here...
});
```

OnClose Event

The **OnClose** event is raised whenever a client is disconnected. The Client is removed from the list and informs the rest of clients about the disconnection.

```
socket.OnClose = () =>
{
// Remove the disconnected client from the list.
clients.Remove(socket);
};
```

OnMessage Event

The **OnMessage** event is raised when a client sends data to the server. Inside this event handler, the incoming message can be transmitted to the clients, or probably select only some of them.

The process is simple. Note that this handler takes a string named **message** as a parameter:

tutorialspoint
SIMPLY EASY LEARNING

```
socket.OnMessage = () =>

{

    // Display the message on the console.

    Console.WriteLine(message);

};
```

Send () Method

The **Send()** method simply transmits the desired message to the specified client. Using Send(), text or binary data can be stored across the clients.

The working of **OnMessage** event is as follows:

```
socket.OnMessage = () =>

{

    foreach (var client in clients)

    {

        // Send the message to everyone!

        // Also, send the client connection's unique identifier in order

        // to recognize who is who.

client.Send(client.ConnectionInfo.Id + " says: " + message);

    }

};
```

API – Definition

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.
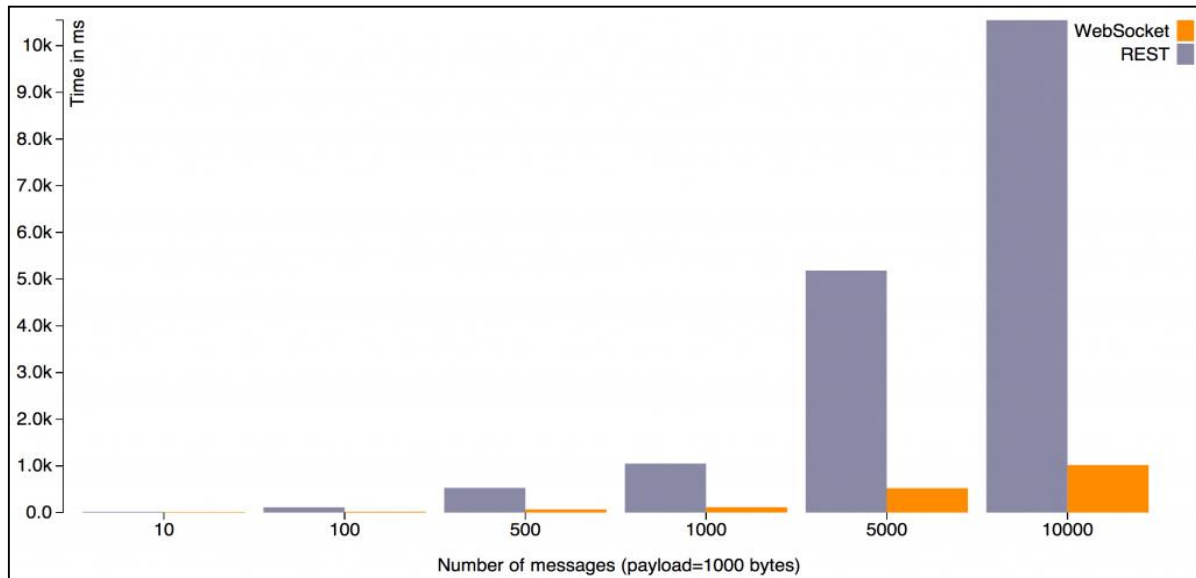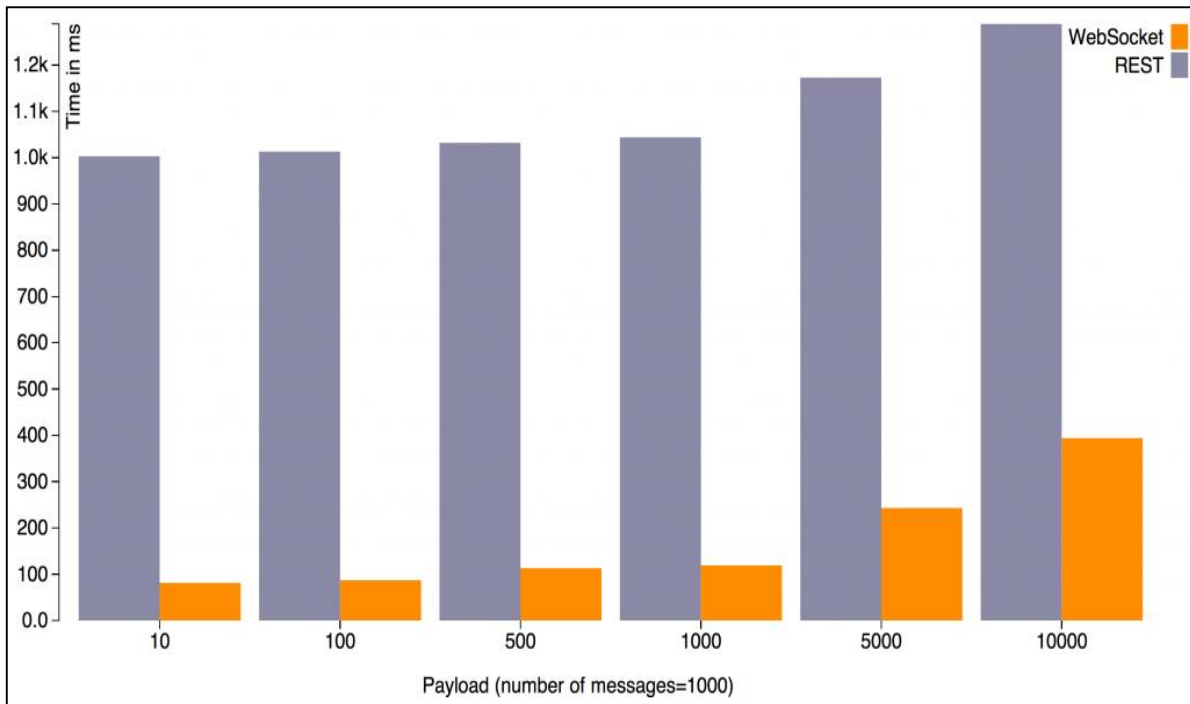

Here is the raw data that feeds this graph:

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.

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

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset=utf-8>

<title>HTML5 Chat</title>

<body>

<section id="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: 10px 0;

}

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: 0 auto;
  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>

  <form onsubmit="addMessage(); return false;">

    <input type="text" id="chat" placeholder="type and press enter to chat" />

  </form>

  <p id="status">Not connected</p>

  <p>Users connected: <span id="connected">0</span></p>

  <ul id="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 (typeof String.prototype.startsWith != "function")

        {

            String.prototype.startsWith = function (str)
```

```
        {
            return this.indexOf(str) == 0;
        };
    }
        window.addEventListener("load", onLoad, false);
}
function onLoad()
{
    var wsUri = "ws://127.0.0.1:7777";
    websocket = new WebSocket(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) };
}
function onOpen(evt)
{
    state.className = "success";
    state.innerHTML = "Connected to server";
}
function onClose(evt)
{
    state.className = "fail";
    state.innerHTML = "Not connected";
    connected.innerHTML = "0";
}
function onMessage(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;
```

```
        }
        else if (message.startsWith("connected:"))

        {
            message = message.slice("connected:".length);

            connected.innerHTML = message;

        }
    }
    function onError(evt)
    {
        state.className = "fail";

        state.innerHTML = "Communication error";

    }
    function addMessage()
    {
        var message = chat.value;

        chat.value = "";

        websocket.send(message);

    }
</script>
</section>
</body>
</html>
```

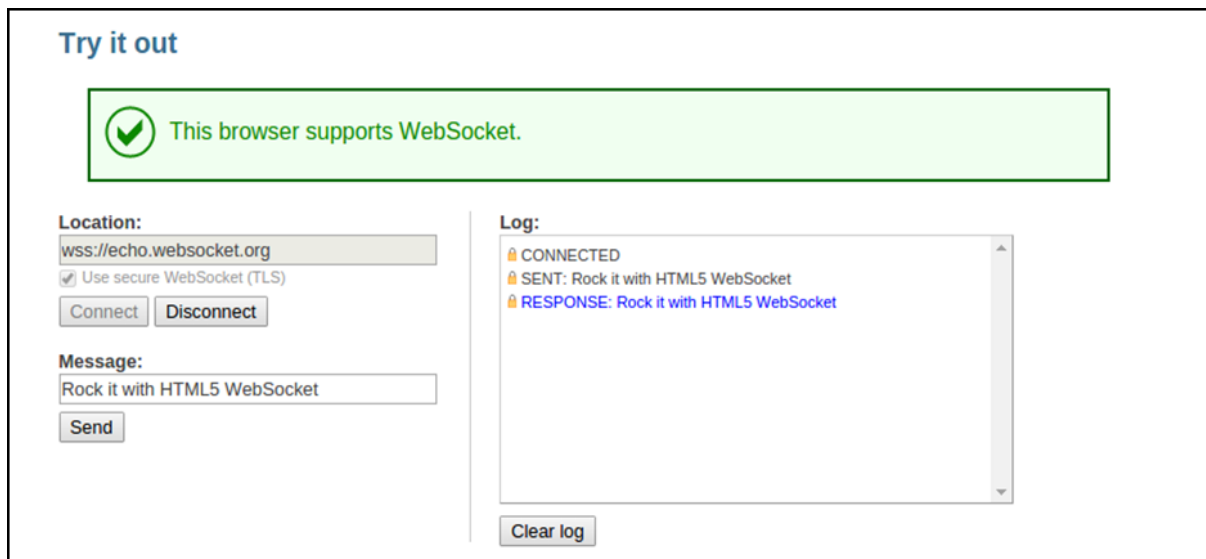The key features and the output of the chat application are discussed below:

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.
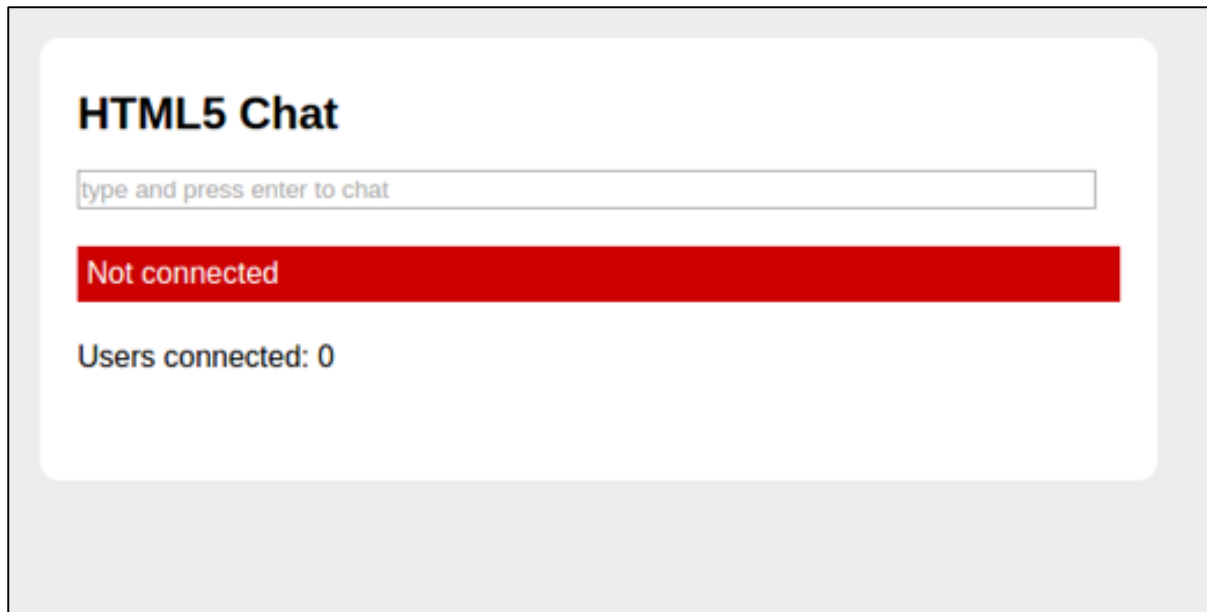
The output of a successful chat communication is shown below.

Unit 4: Web Sockets – Implementation

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);
};
```

Communicating with the Server

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
var img = canvas_context.getImageData(0, 0, 400, 320);
var binary = new Uint8Array(img.data.length);
for (var i = 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.

14.   Web Sockets – 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

Let us understand this with the help of an example.

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.

15.   Web Sockets – Mobile APP

WebSocket, as the name implies, is something that uses the web. The web is usually interwoven with browser pages because that are the primary means of displaying data online. However, non-browser programs too, use online data transmission.

The release of the iPhone (initially) and the iPad (later) introduced a brand new world of web interconnectivity without necessarily using a web browser. Instead, the new smartphone and tablet devices utilized the power of native apps to offer a unique user experience.

Why Mobile Matters?

Currently, there are one billion active smartphones out there. That is, millions of potential customers for your applications. These people use their mobile phone to accomplish daily tasks, surf the internet, communicate, or shop.

Smartphones have become synonymous to apps. Nowadays, there is an app for any usage, a user can think of. Most of the apps connect to the internet in order to retrieve data, make transactions, gather news, and so on.

It would be great to use the existing WebSocket knowledge and develop a WebSocket client running natively on a smartphone or tablet device.

Native Mobile App Vs Mobile Website

Well, this is a common conflict and as usual, the answer depends on the needs of the target audience. If a user is familiar with the modern design trends, designing a website that is responsive and mobile friendly is now a must. However, the end user must be sure that the content, which is what really matters, is equally accessible via a smartphone, as it is via a classic desktop browser.

Definitely, a WebSocket web app will run on any HTML5-compliant browser, including mobile browsers such as Safari for iOS and Chrome for mobile. Therefore, there are no worries about compatibility issues with smartphones.

Prerequisites

In order to develop a smartphone app, installation of development tools and SDKs are required.

Platform	Targets	SDK URL
iOS	iPhone, iPad	`https://developer.apple.com/devcenter/ios/`
Android	Android phones & tablets	`http://developer.android.com/sdk/`
Windows	Windows Phone, Windows 8	`http://developer.windowsphone.com/ develop` & `http://msdn.microsoft.com/ windows/apps`

WebSockets can act as a universal hub for transmitting messages between connected mobile and tablet clients. We can implement a native iOS application, which communicates with a WebSocket server just like the HTML5 JavaScript client.

tutorialspoint
SIMPLYEASYLEARNING