Introduction to Websockets

# About

There are many methods of communication between a server and client. A typical method is by creating a RESTful API using HTTP protocol. This allows the client to send requests to the server to retrieve information from or to update the database. However, this connection is limited because it one-directional. Due to the request-response nature of HTTP requests, there are many limitations. These limitations can be somewhat circumvented using polling and frequent refreshes; however, this introduces a lot of overhead and results in the client waiting for responses. Websockets are a good solution to this problem. Websockets provide a **bi-directional, full-duplex, persistent** **connection** between a client and a server. Once the connection has been established the connection remains open until the client or server decides to close the connection.

Let’s take a look at how they function and how we can use them.

The remaining document is organized in the following manner:

1. A brief overview of some concepts and terms.
2. The dependencies and requirements
3. A few examples of Websockets in action

# Concepts/Terms

## Why/When to use Websockets

As mentioned above, Websockets provide a bi-directional, full-duplex, persistent connection between a client and a server. This allows for real-time communication between the client and the server with little overhead. If you were to try to recreate the same behavior with HTTP requests, you would need to implement polling either synchronously/asynchronously, which makes the situation quite complex.

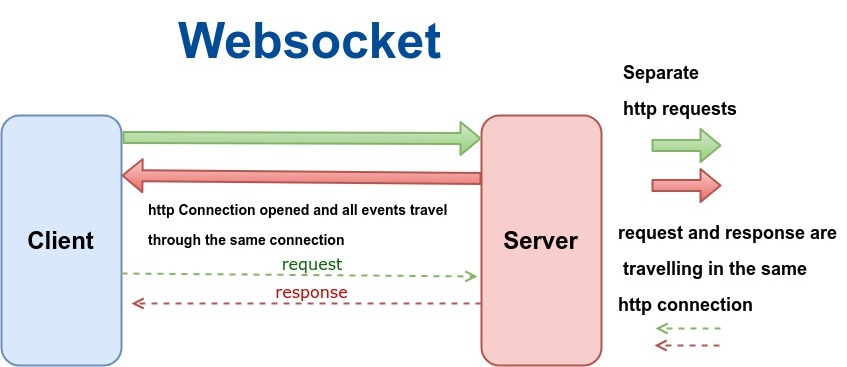
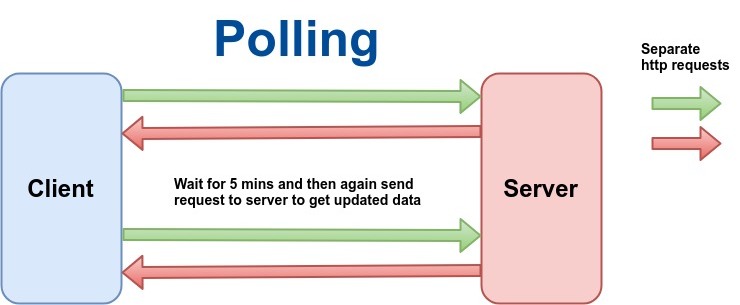


Figure 1 Polling with requests vs Websocket connection

## Bi-directional & Full duplex

Http-requests are bi-direction and half-duplex, which means while the communication goes both ways, it can only be in one direction at a time. This is very similar to how walkie-talkies function. In contrast, Websockets are full-duplex, which means they allow both sending and receiving to occur at the same time. This is similar to a phone call in which you can talk while the other person is also talking.

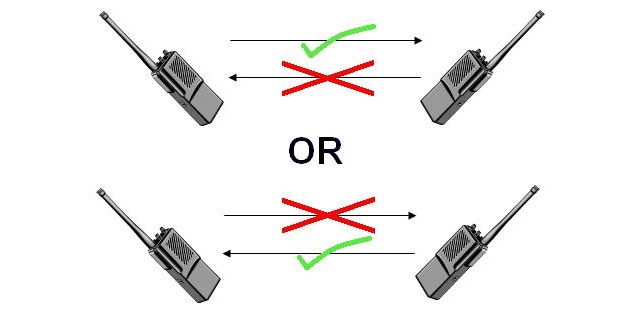


Figure 2 Half-Duplex vs Full-Duplex

## Persistent connection

Let’s say the server has received a request from a client and the server wants to update all the other clients of the changes made by the request. There is no way to do this using HTTP requests. Websockets’ establish a connection that is not disconnected until one side decides to disconnect. Websockets have an initial handshake which is simply an HTTP request (Figure 1). If the request is accepted the connection is upgraded to a Websocket connection. This property allows the server to push information to the client without the client actively requesting the information. The server can maintain a list of clients which it can use to send updated information to all clients. This is why real-time communication is possible through Websockets.

## Summary of HTTP requests vs Websockets

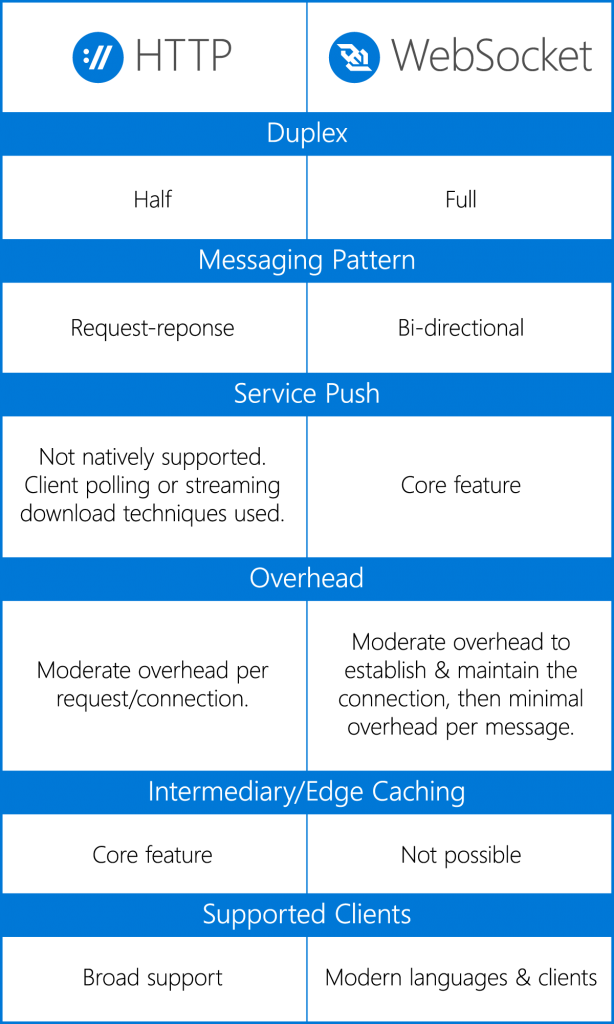


Table 1 Features of HTTP requests vs Websockets

# Websocket Basic Example

## Introduction

In this example, we have a basic JS websocket client implementation connected to a websocket server that simply echoes what we send to it. This application is similar to a chat application. Connecting to an echo server is a great way to test your websocket client before you start connecting to your own server. It allows you to see if you have formatted the message correctly and whether your client successfully connects to an endpoint.

Code link *(may need to copy/paste link)*: <https://codepen.io/AniTang/full/VwLQGVx>

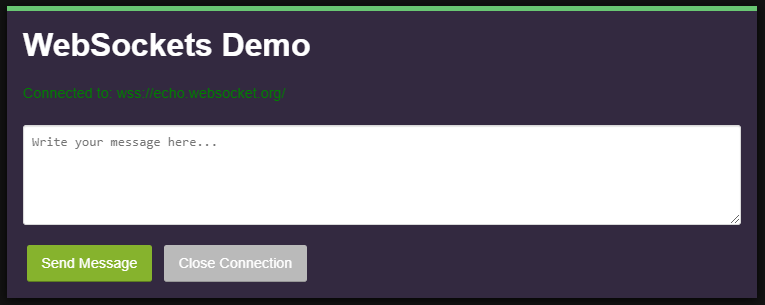


Figure 3 Working Example of Websocket in JS

While the code may not be directly applicable to your project, this example aptly demonstrates how a websocket client works. Let’s take a closer look at the JS script. Change views to editor view.

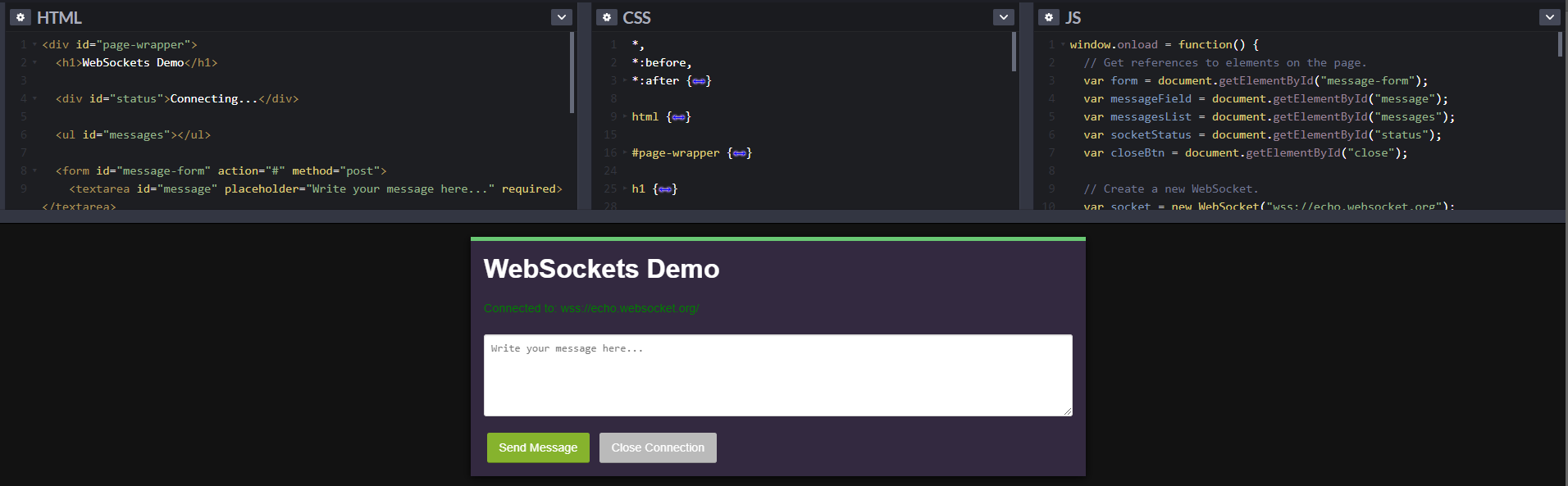


Figure 4 Editor View

The script implements four websocket methods: *onOpen()*, *onMessage()*, *onClose()*,and *onError().* These methods typically need to be implemented when using websockets in Java as well. If you are using a library that provides a base implementation, then you may not need to implement these methods.

## Websocket Methods



## OnOpen vs Initial Connection

When you try to connect to a websocket endpoint, the initial request sent is an HTTP request with “upgrade” headers. This is NOT what the *onOpen()* method does. *OnOpen()* is called when the endpoint accepts the request and sends an upgrade response. Therefore, this method is called after the connection has been opened. This method is a good place to initialize variables and start your processes.

## OnMessage

The *onMessage()* method is called when the socket receives a message from the endpoint. This method is NOT called when sending a message through the socket. Therefore, you must parse and handle the received messages in this method or call other methods that will parse the message.

## OnClose and OnError

*OnClose()* is called when the websocket client disconnects from the endpoint or the server disconnects the client. *OnClose()* is also typically called after *onError()* in most websocket implementations. *OnError()* is called when the socket experiences an error AFTER the connection has been opened. This method is not called if the websocket connection fails the initial connection.

## Sending Messages

You may have noticed that a key method/feature is missing: sending messages. Don’t worry, this is much simpler than you think. To send a message, you simply need to call *socket.send(InsertMessageHere)*. The sending of the message is usually handled by the specific websocket library you are using. The type of message is also dictated by the library. Most websocket libraries support JSON objects to be sent. If you are using barebones sockets then you may need to send raw bytes.

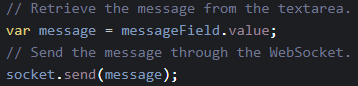


Figure 5 Socket send() Method

## Client vs Server methods

While it may seem the code above is targeted specifically towards client-side, this is not the case. Depending on the websocket library you are using, the server-side may also need to implement the exact same methods on the backend. However, you will most likely handle messages, errors and connections differently. The biggest change on the server-side will be the *onOpen()*. You will most likely want to store all clients that connect to the server in some data structure to easily access the client and send updates to them. Remember, the sockets/clients are also just objects so you can treat them as such. We will go into more implementation specific details in the next example where we will be looking at a Spring Boot server with websockets.

## Higher Level vs Lower Level Websocket Library

Depending on the library used, the above methods may or may not be implemented. Websockets are much lower level than HTTP requests so they can be a little tedious to work with as they require some setup before use. In this situation, you can use a library which implements a basic websocket and all you have to do is configure the type of messages sent and implement a few message handlers. An example of this is STOMP over websockets. STOMP is a text-oriented messaging protocol which is also designed to run on TCP like websockets and is meant to be able to connect to clients across languages and architectures. It simplifies the connection but requires a little bit more setup with message handling. Lower level libraries like Javax websockets will require you to implement the methods mentioned earlier, however, they allow more control and easier access to the message.

# Acknowledgements

**If some parts of this document are not clear, or you feel that there could be more examples – please let us know on Piazza.**