WebSocket

Introduction

WebSocket as a Web Communication Protocol: WebSocket is a computer communications protocol, providing simultaneous two-way communication channels over a single Transmission Control Protocol (TCP) connection.

Bi-directional vs. Uni-directional Channels: WebSocket enables real-time bidirectional communication between the client and server, allowing data to flow in both directions simultaneously (full-duplex). This is in contrast to HTTP, which follows a unidirectional request-response pattern where the client makes a request and the server responds. Once the response is delivered, the HTTP connection is closed, and a new connection must be established for each new request.

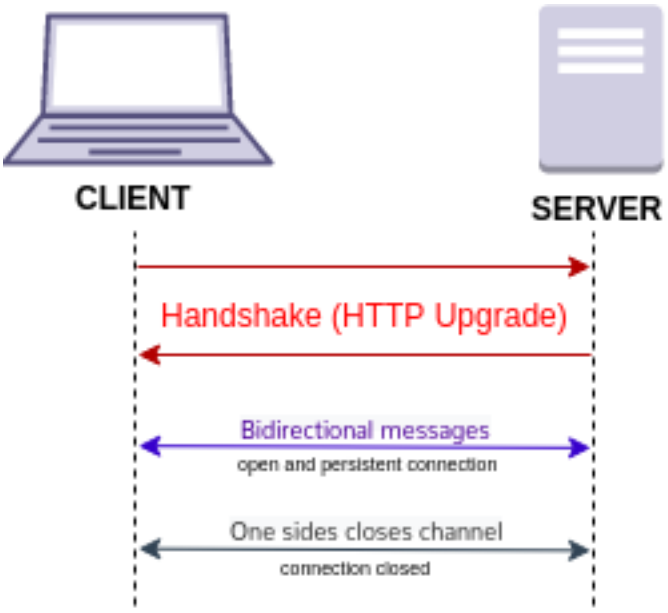
Initial Connection Establishment: The WebSocket connection begins with an HTTP request/response. During this "handshake" phase, the client sends an HTTP request to upgrade to a WebSocket connection. If the server supports WebSocket, it responds affirmatively, and the connection is then "upgraded" from HTTP to WebSocket, allowing for the aforementioned bidirectional communication.

Two-Way Communication Channel: WebSocket facilitates a continuous two-way communication channel between the client and server, enabling scenarios where the server can push content to the client without a specific request from the client. This capability is particularly beneficial for real-time applications where immediate data transfer is crucial.

Use Cases for WebSocket: WebSocket is well-suited for applications requiring live interaction and real-time updates. Common applications include chat apps, collaboration tools (like Google Docs), multiplayer games, live dashboards, and live streaming applications. These applications benefit from WebSocket's ability to provide instant communication and updates.

WebSocket APIs and Libraries: There are numerous JavaScript libraries that facilitate interaction with the WebSocket API, enhancing its functionality and ease of use in various application scenarios. Libraries like Socket.IO

How a WebSocket app works



1. **WebSocket Connection Initiation**: The process begins with the client initiating a WebSocket connection. This is done by sending an HTTP handshake request to the server with an “Upgrade” header, signaling the desire to start using the WebSocket protocol instead of the standard HTTP protocol. This step is crucial as it sets the stage for a different kind of communication compared to the typical HTTP request-response model.
2. **Establishing the WebSocket Connection**: Once the server agrees to the upgrade request, the HTTP connection is transformed into a WebSocket connection. This upgraded connection is kept alive, unlike traditional HTTP connections that close after each request-response cycle. This persistent connection is what allows for real-time, bidirectional communication between the client and server.
3. **Data Transmission over WebSocket**: After the successful establishment of the WebSocket connection, the client and server can exchange data in real-time using the WebSocket API. The data can be transmitted in either direction at any time, without the need for a new request for each piece of data. This is a significant departure from the standard HTTP protocol, where data flows only in response to a client's request. The real-time nature of this communication is particularly useful for applications that require immediate data updates, such as chat applications, live feeds, or interactive gaming.
4. **Maintaining and Monitoring the Connection**: Both the client and server can send "ping" and "pong" messages to each other to ensure that the connection is still alive

and to detect any connection issues. This helps in maintaining a stable and responsive communication channel.

5. **Closing the WebSocket Connection**: The WebSocket connection can be terminated by either the client or the server. This can be a deliberate action, or the connection might close due to network errors or timeouts. The flexibility of closing the connection from either end is an aspect of the protocol's design to accommodate different application needs and scenarios.

Http vs WebSocket

Aspect	HTTP	WebSocket
Communication Model	It operates on a request-response protocol, meaning the client initiates communication by sending a request and the server responds. This protocol is unidirectional, as each communication is either a request or a response. After the response, the connection is terminated, and for a new request, a new connection must be established.	It operates on a request-response protocol, meaning the client initiates communication by sending a request and the server responds. This protocol is unidirectional, as each communication is either a request or a response. After the response, the connection is terminated, and for a new request, a new connection must be established.
Connection Persistence	The connection in HTTP is not persistent. It is established for each request and closed after the response is delivered. This results in a new connection needing to be made for each HTTP request.	Once established, the WebSocket connection remains open, allowing for continuous communication between the client and server until either party decides to close it. This persistent connection makes communication more efficient by eliminating the need for repeated handshake requests.
Statefulness	HTTP is a stateless protocol, meaning that	Being a full-duplex protocol, WebSocket can

	each request from a client to server must contain all the information necessary for the server to fulfill it. This stateless design is scalable but can limit certain functionalities, like user sessions or shopping carts, which require persistent state information.	maintain a stateful connection, which is beneficial for applications requiring continuous data exchange or real-time updates.
Usage Scenarios	Ideal for web browsing, downloading images or videos, and making asynchronous API requests. It's used for applications where the client typically initiates communication and where statelessness is either acceptable or preferred	Preferred for real-time applications, such as online gaming, live chat applications, and collaborative platforms. It's used in scenarios where continuous data exchange is required without the overhead of repeatedly establishing

Socket.io

Socket.IO is an event-driven JavaScript library designed for real-time web applications. It enables real-time, bi-directional communication between web clients and servers and consists of two main components: a client that runs in the browser and a server that runs on Node.js. Socket.IO abstracts the underlying transport protocols, such as WebSocket, while providing a simple and unified API for developers to use. It's open-source and supports multiple languages, including Deno (JavaScript), C++, Java, and Swift for server-side implementations.

Brief History

Socket.IO was created to address the limitations and complexities associated with real-time communication in web applications, particularly those that relied on traditional technologies like AJAX. Before Socket.IO, creating real-time applications like chat systems required constant polling of the server for changes, which was inefficient and

slow. Socket.IO provided a more efficient solution for real-time data transfer and has since become a standard for such applications.

Relation to WebSocket

While Socket.IO primarily uses the WebSocket protocol, it's not limited to it. Socket.IO is a custom real-time transfer protocol implementation built on top of WebSocket and other real-time protocols. It provides a wrapper for WebSockets but includes additional features like automatic fallback to HTTP long-polling, broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O. This makes Socket.IO more versatile and robust than using WebSocket alone, especially in environments where WebSocket is not supported or is blocked by firewalls.

Simplest Usage of Socket.IO

- In the following example, The client-side JavaScript establishes a connection with the Socket.IO server, sends chat messages, and listens for incoming messages.
- The server-side JavaScript listens for incoming chat messages and then broadcasts them to all connected clients.

	
Client side	Server side

Hands-On

Explore a simple chat app built using Socket.io

- Go to the following chat app project on GitHub
- Download the archive.zip
- Open the code in your local code editor
- Go over the code and read all the comments to understand how it works
- Run the app and test it

Building your own chat app step by step

Go to https://socket.io/docs/v4/tutorial/introduction and follow the tutorial until step #5

Challenges

- Integrate socket.io into your own website (live chat support, multiplayer game, etc)
- Complete the tutorial until step #9 to know how to deal with interruptions, scaling and advanced concepts.

References

- https://ably.com/topic/websockets-vs-http
- https://oxylabs.io/blog/websocket-vs-http
- https://www.wallarm.com/what/websocket-vs-http-how-are-these-2-different
- https://en.wikipedia.org/wiki/WebSocket
- https://blog.logrocket.com/top-websocket-libraries-nodejs-2022/#:~:text=WebSockets%20provide%20an%20open%20connection,updates%2C%20to%20name%20a%20few
- https://www.knowledgehut.com/blog/web-development/what-is-websocket
- https://ably.com/topic/websockets
- https://socket.io/docs/v4/how-it-works/
- https://dev.to/cglikpo/getting-started-with-socket-io-7m4#:~:text=,broadcast%20them%20to%20other%20clients
-