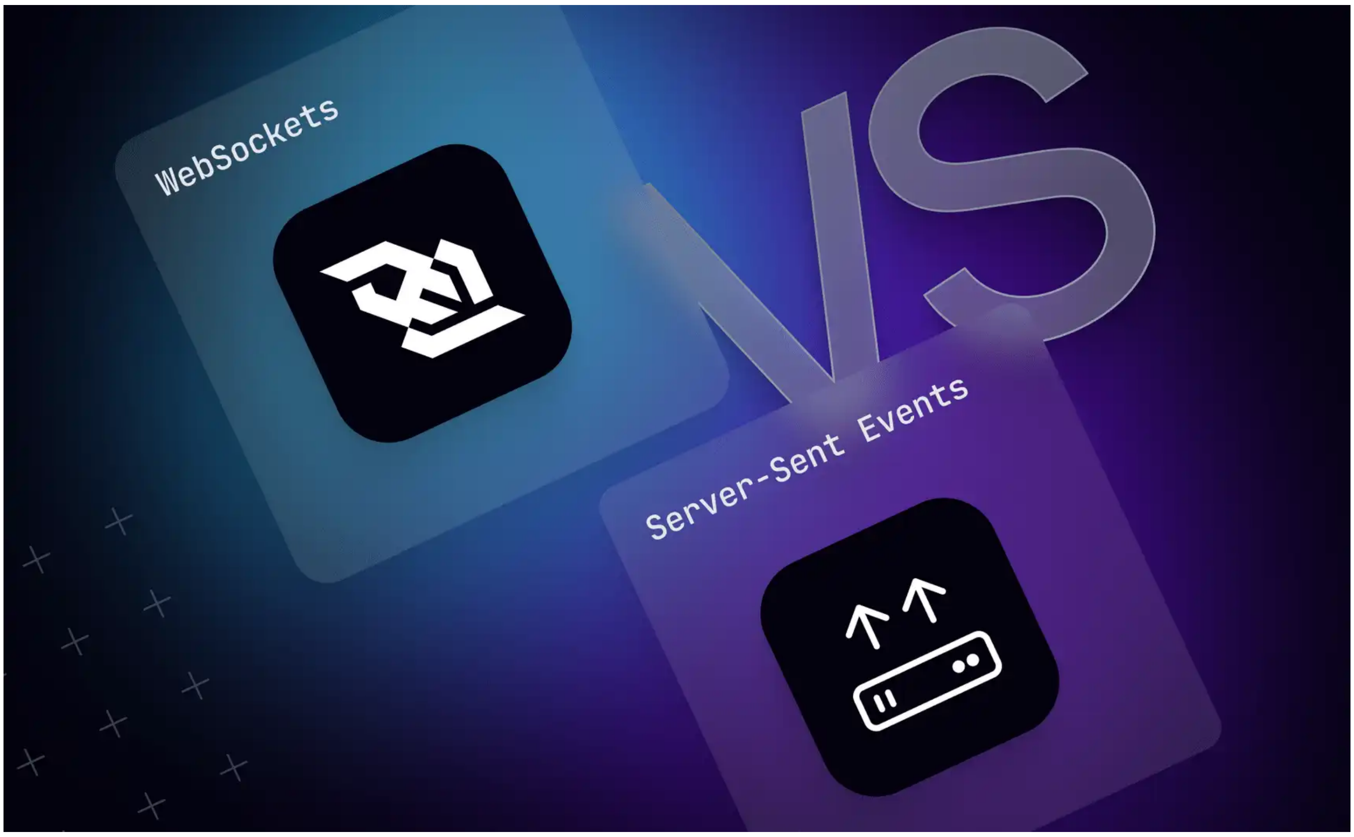
WebSockets vs Server-Sent Events: Key differences and which to use in 2024



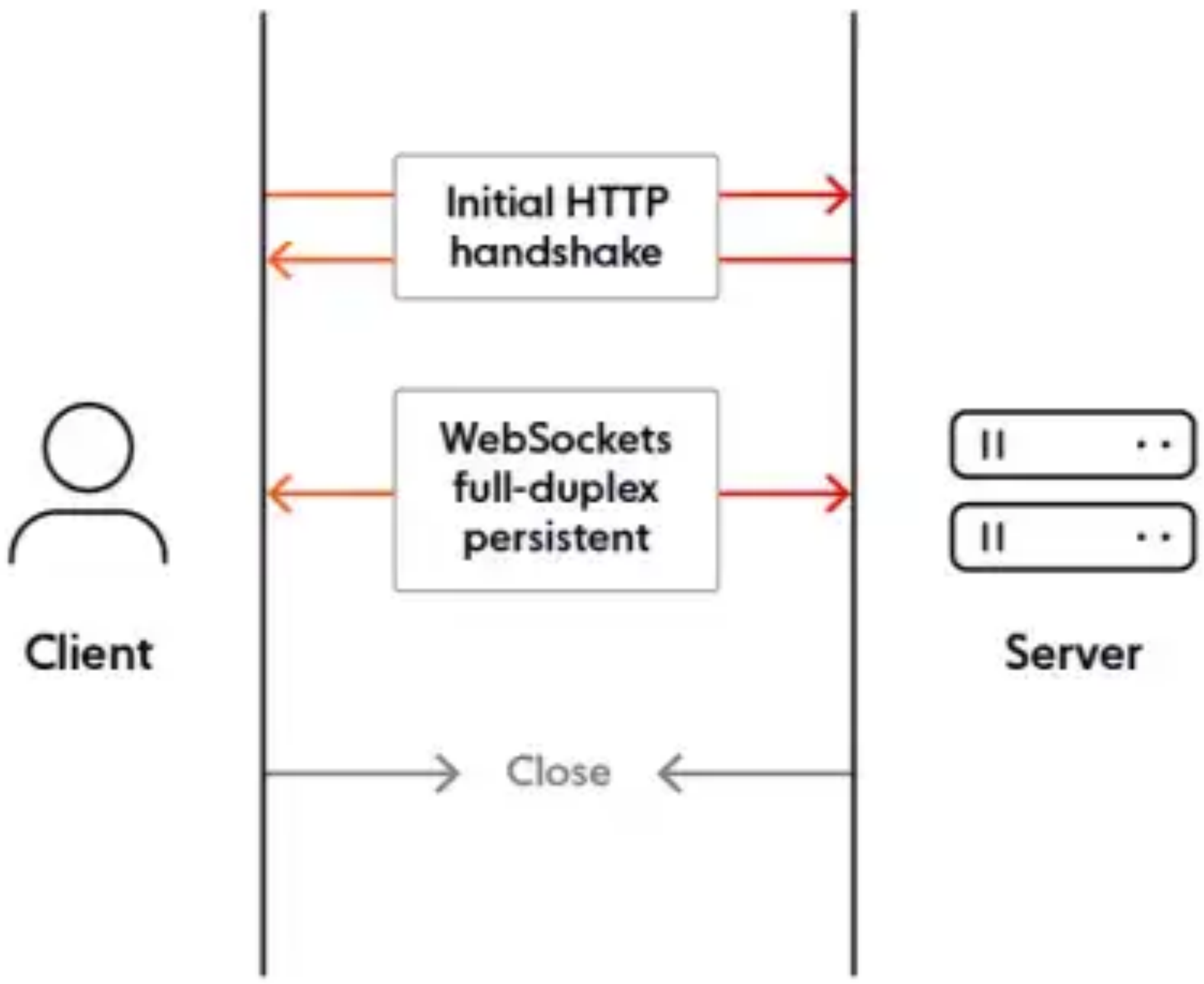
WebSockets and Server-Sent Events are commonly used in realtime applications where quick and efficient data transfer is a critical requirement. The expectations of realtime experiences in applications has only grown with time, with improving technology and understanding of what is possible.

This article compares two popular realtime protocols — the WebSockets and Server-Sent Event APIs. Below you'll learn what each is capable of, their pros and cons, and when to use them.

What are WebSockets?

WebSockets are a thin transport layer built on top of a device's TCP/IP stack, which provides full-duplex, low-latency, event-driven connections between the server and the browser. This is ideal for realtime apps since, after the initial HTTP handshake, a single WebSocket connection can handle all of the messages for a single session, without any further handshakes. When the session finishes, the connection should be closed as part of the clean-up.

READ MORE: Learn how WebSockets work



Summary of key WebSockets features

- Uses a custom ws protocol to transport messages, which works at a lower level than HTTP.
- Connection is two-way, so WebSockets are useful for apps that

require data to be read from and written to the server, such as chat apps or multiplayer games.
- It can be complex to implement WebSocket from the ground up, but there are a wide variety of libraries available to facilitate this.
- Event-based; no polling required to retrieve messages, helping to reduce latencies.
- RFC 6455 – The WebSocket Protocol was published to the IETF website in 2011, and all major browsers now support it. See the WebSockets browser support table on MDN for more details.

WebSockets advantages and disadvantages

WebSockets advantages:

- **Fallback to HTTP:** If you need to fall back to HTTP to get over the shortcomings of WebSockets mentioned below, this is possible to achieve using popular WebSockets-based libraries like Socket.IO, which have such fallbacks built-in.
- **Resource efficiency:** Due to being a low-level protocol, a single WebSocket connection can handle a high bandwidth on a single connection. WebSockets do not use `XMLHttpRequest`, and headers are not sent every time we need to get more information from the server. This minimizes the expensive data loads sent to the server.
- **High performance and low latency:** Unlike traditional HTTP protocols, which initiate a new request-response cycle for each data transfer, WebSockets maintain a persistent connection between the client and server. This reduces latency and overhead, as fewer headers and handshakes are required with each message. Consequently, WebSockets offer significant performance advantages for low-latency use cases like chat and live data updates.
- **Bi-directional communication in realtime:** Because WebSocket provides a full-duplex, bi-directional communication channel, the server can send messages to the client, and both can send messages at the same time. This makes two-way, multi-user realtime apps highly performant.

- **Data format flexibility:** WebSockets can transmit binary data and UTF-8 meaning that apps can support sending plain text and binary formats such as images and video.

WebSockets disadvantages:

- **Firewall blocking:** Some enterprise firewalls with packet inspection have trouble dealing with WebSockets (notably SophosXG Firewall, WatchGuard, and McAfee Web Gateway).
- **No built-in support for reconnection:** When a WebSocket connection is closed (e.g. due to network issues), the client does not try to reconnect to the server, which means you'll need to write extra code to poll the server, re-establishing the connection when it is available again. Alternatively, you could use Server-Sent Events, or a library with reconnection support like Socket.IO.

READ MORE: The pros and cons of WebSockets

Getting started with WebSockets

To get started with WebSockets, it is common to use a library to implement a server using a language like Node.js, and then connect to that server from a client using the client-side WebSockets API.

There are two primary classes of WebSocket libraries:

- Those that implement the protocol and leave the rest to the developer, like ws.
- Those that build on top of the protocol with various additional features commonly required by realtime messaging applications, such as restoring lost connections, pub/sub, channels, authentication, and authorization. Examples include Socket.IO and SockJS.

Although WebSocket libraries that build on top of the protocol have lots to offer from a feature perspective, it is important to note that they often require their own libraries to be used on the client-side - rather than just using the raw WebSocket API provided by the browser. Once you integrate the solution, you are relatively locked in. Make sure you consider whether the library is a good fit for your long-term plans and scale up-front.

In this section, you'll get started with WebSockets by creating a WebSocket server using a library that sits in the first category — ws: a simple and reliable solution for Node.js. We'll then show you how to connect to that server instance in the browser using the native WebSocket API.

Creating a WebSocket server with ws

The following code snippet creates a simple Node.js WebSocket server:

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    console.log('received: %s', message);
  });

  ws.send('something');
});
```

After requiring the library and creating a new WebSocket server, we attach a connection event listener to the server, to listen for connections. Once a connection occurs, a message event listener is set up to listen for messages, and send a message back to the connected client.

Find more documentation and examples on the ws npm page.

Using WebSockets in the browser

To connect to a WebSocket server from a browser, you'll need something like this:

```
const ws = new WebSocket('ws://example.org');

ws.addEventListener('open', () => {
  // Send a message to the WebSocket server
  ws.send('Hello!');
});

ws.addEventListener('message', event => {
  // The `event` object is a typical DOM event object, and the message data sent
  // by the server is stored in the `data` property
  console.log('Received:', event.data);
});
```

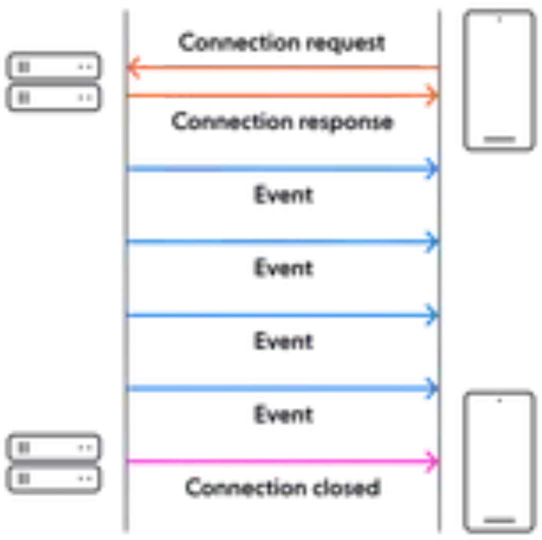
After connecting to the server via its URL (note the use of the ws protocol), we set up event listeners to listen for the completion of the HTTP handshake before sending any messages, and to react to messages received from the server.

While the WebSocket class is straightforward and easy to use, it is just a basic building block. Additional features like shared messaging channels and automatic reconnection need to be separately implemented. This is why there are many client-side libraries in existence.

For more documentation, see The WebSocket API on MDN.

What are Server-Sent Events?

Server-Sent Events (SSE) are based on Server-Sent DOM Events. Browsers can subscribe to a stream of events generated by a server using the EventSource interface, receiving updates whenever a new event occurs. EventSource accepts an HTTP event stream connection from a specific URL and keeps the connection open while retrieving available data. Server-sent events are pushed (rather than pulled, or requested) from a server to a browser.



Server-Sent Events is a standard describing how servers can maintain data transmission to clients after an initial client connection has been established. It provides a memory-efficient implementation of XHR streaming. Unlike a raw XHR connection, which buffers the entire received response until the connection drops, an SSE connection can discard processed messages without accumulating all of them in memory.

Summary of key Server-Sent Events features

- Uses XHR streaming to transport messages over HTTP.
- Connection is one-way, so SSEs are useful for apps that only require reading data from the server, such as live stock or news tickers.
- Less complex to implement SSE-based connections, but not many related libraries are available.
- Event-based; no polling required to intercept messages.
- Server-Sent Events are specified in the HTML specification and have been supported by all major browsers for a number of years. See the EventSource browser support table on MDN for more details.

Server-Sent Events advantages:

- **Polyfillable:** Server-Sent Events can be poly-filled with JavaScript in browsers that do not support it yet. This is useful for backward compatibility because you can rely on the existing implementation

rather than having to write an alternative.
- **Built-in support for reconnection:** Server-Sent Event connections will reestablish a connection after it is lost, meaning less code to write to achieve an essential behavior.
- **No firewall blocking:** SSEs have no trouble with corporate firewalls doing packet inspection, which is important for supporting apps in enterprise settings.

Server-Sent Events disadvantages:

- **Data format limitations.** Server-Sent Events are limited to transporting UTF-8 messages; binary data is not supported.
- **Limited concurrent connections.** You can only have six concurrent open SSE connections per browser at any one time. This can be especially painful when you want to open multiple tabs with SSE connections. See 'Server-Sent Events and browser limits' for more information and workaround suggestions.
- **Mono-directional.** SSE can only send messages from server to client. While this is useful for creating read-only realtime apps like stock tickers, it is limiting for many other types of realtime app.

Getting started with Server-Sent Events

In this section you'll look at how to get started on experimenting with SSEs, implementing a simple server with `express-sse` to keep things clean, and connecting to it from a browser using EventSource.

Creating an SSE server with express-sse

The following code snippet creates a simple Node.js SSE server:

```
var SSE = require('express-sse');
var sse = new SSE();
app.get('/stream', sse.init);
// we can always call sse.send from anywhere the sse variable is available, and see the r
```

```
let content = 'Test data at ' + JSON.stringify(Date.now());
sse.send(content);
```

After creating a stream with sse.init, a simple message is sent down the stream using sse.send().

For more documentation and examples, see the sse-express npm page.

Using SSE in the browser

To connect to a SSE server from a browser, you'll need something like this:

```
const evtSource = new EventSource("/stream");

eventSource.addEventListener('message', event => {
    console.log(event.data);
}
```

EventSource is passed the stream URL to connect to it. Once connected, we set up a message event listener on the event source to handle received messages. For our simple example, we will log the message to the console.

For more documentation, see Server-sent events on MDN.

What is the difference between WebSockets and Server-Sent Events?

The following table provides a quick summary of the key differences between WebSockets and Server-Sent Events.

WebSockets	Server-Sent Events
Two-way message transmission	One-way message transmission (server to client)
Supports binary and UTF-8 data transmission	Supports UTF-8 data transmission only

Supports a large number of connections per browser	Supports a limited number of connections per browser (six)
Can't be polyfilled using JavaScript; you need to fall back to basic HTTP messages	Can be polyfilled using JavaScript
Some enterprise firewalls with packet inspection have trouble dealing with WebSockets	No blocking by enterprise firewalls

Should I use WebSockets or Server-Sent Events?

Which technology to use depends on your use case. In this section, we will look at suitable use cases for both.

When to use WebSockets

WebSockets are more complex and demanding than SSE, and require a bit of developer input up front. For this investment, you gain a full-duplex TCP connection that is useful for a wide range of application scenarios. For example, WebSockets are preferable for use cases such as multiplayer collaboration and chat apps. In addition, WebSockets often perform better than traditional HTTP protocols in low latency and high-bandwidth connections. While SSE + AJAX can technically be used to achieve these use cases, this can result in desynchronized communication and more work than might be required with WebSockets.

READ MORE: When to use WebSockets

Why use Server-Sent Events over WebSockets?

Server-Sent Events is a good alternative to WebSockets for simple realtime use cases that only require one-way communication (from server to client). Examples include read-only realtime apps like stock tickers, or news updates. However, bear in mind that you can only send UTF-8 data over SSE (binary data is not supported).

Server-Sent Events vs WebSockets - which is

better?

SSE is a simpler solution, but it isn't extensible: if your web application requirements were to change, the likelihood is it would eventually need to be refactored using WebSockets. Although WebSocket technology presents more upfront work, it's a more versatile and extensible framework, so a better option for complex applications that are likely to add new features over time.

Ably is an alternative to using raw WebSockets with robust pub/sub messaging, over 25 SDKs targeting every major language and development platform, with dedicated expert support available. With both WebSocket and REST libraries available, you can integrate with ease.

Give Ably a try — sign up for a free account and begin exploring by working through our Quickstart guide.

Further reading on WebSockets

- NEW: WebRTC vs WebSocket
- NEW: Socket.IO vs. WebSocket
- Can WebTransport Replace WebSockets?
- WebSockets vs Long Polling
- How to build a serverless WebSockets platform

Join the Ably newsletter today

1000s of industry pioneers trust Ably for monthly insights on the realtime data economy.