

WebSocket API Lab — Browser Side

Course: Advanced Web Technologies

Duration: ~2 hours

Learning Outcomes

- Open a WebSocket connection from your browser using the WebSocket API.
- Send and receive real-time messages without refreshing the page.
- Listen for connection events (open, message, error, close).
- Explain the WebSocket connection lifecycle.

What You'll Need

- A modern web browser (Chrome, Edge, Firefox).
- A text editor (VS Code or similar).
- Internet connection (we'll use a public WebSocket echo server).

Public test server: `wss://echo.websocket.org`

Setting Up the Page

Step 1. Create project folder: `index.html`, `script.js`, and `styles.css`.

Step 2. HTML (`index.html`):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>WebSocket Lab</title>
    <link rel="stylesheet" href="styles.css" />
    <script defer src="script.js"></script>
  </head>
  <body>
    <h1>🌿 WebSocket Demo</h1>

    <div id="status" class="disconnected" aria-live="polite">
      Status: Disconnected ❌
    </div>
```

```

<div class="controls">
  <button id="openBtn">Open Connection</button>
  <input
    type="text"
    id="messageInput"
    placeholder="Type your message..."
    aria-label="Message to send"
  />
  <button id="sendBtn" disabled>Send</button>
  <button id="closeBtn" disabled>Close Connection</button>
</div>

<pre id="output" aria-live="polite" aria-atomic="false"></pre>
</body>
</html>

```

Step 3. JavaScript (script.js):

```

// URL for public echo server (secure)
const WS_URL = "wss://echo.websocket.org";

// DOM refs
const statusEl = document.getElementById("status");
const openBtn = document.getElementById("openBtn");
const sendBtn = document.getElementById("sendBtn");
const closeBtn = document.getElementById("closeBtn");
const input = document.getElementById("messageInput");
const output = document.getElementById("output");

let socket = null;

/** UI helpers */
function setStatus(text, cls) {
  statusEl.textContent = text;
  statusEl.className = cls;
}

function setButtons({ connected, connecting }) {
  // Send only when connected
  sendBtn.disabled = !connected;
  // Close only when connecting or connected? Typically only connected.
  closeBtn.disabled = !connected;

  // Open disabled while connected or connecting
  openBtn.disabled = connected || connecting;
}

/** Append a line to the output and autoscroll */

```



```

function logLine(line) {
    output.textContent += (output.textContent ? "\n" : "") + line;
    output.scrollTop = output.scrollHeight;
}


/** Open the WebSocket connection */
function openConnection() {
    if (
        socket &&
        (socket.readyState === WebSocket.OPEN ||
         socket.readyState === WebSocket.CONNECTING)
    ) {
        return; // already open/connecting
    }



    setStatus("Status: Connecting...", "connecting");
    setButtons({ connected: false, connecting: true });

    socket = new WebSocket(WS_URL);

    socket.onopen = () => {
        setStatus("Status: Connected ", "connected");
        setButtons({ connected: true, connecting: false });
        logLine(` Connected to " + WS_URL);
        input.focus();
    };

    socket.onmessage = (e) => {
        logLine("Server: " + e.data);
    };

    socket.onerror = (e) => {
        // Browsers often give a generic error object; provide a helpful
        message
        logLine(` WebSocket error (see DevTools console for details)`);
        console.debug("WebSocket error:", e);
    };

    socket.onclose = (evt) => {
        setStatus("Status: Disconnected ", "disconnected");
        setButtons({ connected: false, connecting: false });
        const code = evt.code;
        const reason = evt.reason || "no reason provided";
        logLine(` Connection closed (code ${code}) - ${reason}`);
    };
}

/** Send message if connected */
function sendMessage() {

```

```

const msg = input.value.trim();
if (!msg) return;
if (!socket || socket.readyState !== WebSocket.OPEN) {
  logLine("✗ Cannot send – socket is not open.");
  return;
}
socket.send(msg);
logLine("You: " + msg);
input.value = "";
input.focus();
}

/** Close connection gracefully */
function closeConnection() {
  if (
    socket &&
    (socket.readyState === WebSocket.OPEN ||
     socket.readyState === WebSocket.CONNECTING)
  ) {
    socket.close(1000, "User requested close"); // Normal closure
  }
}

// Wire up UI
openBtn.addEventListener("click", openConnection);
sendBtn.addEventListener("click", sendMessage);
closeBtn.addEventListener("click", closeConnection);

// Send on Enter
input.addEventListener("keydown", (e) => {
  if (e.key === "Enter") {
    e.preventDefault();
    sendMessage();
  }
});

// Initial UI state
setStatus("Status: Disconnected ✗", "disconnected");
setButtons({ connected: false, connecting: false });

```

Step 4. CSS(styles.js):

```

/* WebSocket Lab - styles.css */

:root {
  --bg: #f9fafb;
  --text: #1f2937;
  --primary: #0ea5e9;
  --primary-hover: #0284c7;

```

```
--ok: #16a34a;
--warn: #f59e0b;
--bad: #dc2626;
--panel: #f3f4f6;
}

* {
  box-sizing: border-box;
}

body {
  font-family: system-ui, -apple-system, Segoe UI, Roboto, sans-serif;
  background: var(--bg);
  color: var(--text);
  margin: 40px;
}

h1 {
  text-align: center;
  margin-top: 0;
}

#status {
  font-weight: 700;
  margin: 16px 0 12px;
}

.connected {
  color: var(--ok);
}

.connecting {
  color: var(--warn);
}

.disconnected {
  color: var(--bad);
}

.controls {
  display: grid;
  grid-template-columns: 160px 1fr 100px 160px;
  gap: 10px;
  align-items: center;
  margin-bottom: 14px;
}

input[type="text"] {
  padding: 10px 12px;
  border: 1px solid #d1d5db;
  border-radius: 6px;
}
```

```
    outline: none;
}

input[type="text"]:focus {
    border-color: var(--primary);
    box-shadow: 0 0 0 3px rgba(14, 165, 233, 0.2);
}

button {
    padding: 10px 12px;
    border: none;
    border-radius: 6px;
    background: var(--primary);
    color: white;
    cursor: pointer;
    font-weight: 600;
}

button:hover {
    background: var(--primary-hover);
}

button:disabled {
    background: #9ca3af;
    cursor: not-allowed;
}

#output {
    background: var(--panel);
    border: 1px solid #e5e7eb;
    border-radius: 8px;
    padding: 12px;
    min-height: 240px;
    max-height: 420px;
    overflow-y: auto;
    white-space: pre-wrap;
    font-family: ui-monospace, SFMono-Regular, Menlo, Consolas, monospace;
}
```

Explore and Observe

Use the **Console** to observe how the `readyState` property changes during each stage of a WebSocket connection.

Before Opening the Connection

- Open the Console and type:

```
socket.readyState
```

1. What number do you see? What does that value represent in the WebSocket lifecycle?

- There is a null error, since the socket instance has not been created. However, if the socket instance is created at the start of the page, the number it shows is 3, which represents a closed state.

Immediately After Clicking “Open Connection”

- Type again:

```
socket.readyState
```

2. What value do you see now? Which constant does it match (CONNECTING, OPEN, CLOSING, or CLOSED)?

- The value is now 0. Which means CONNECTING

When the “Status: Connected ” Appears

- Check `socket.readyState` again.

3. What value is displayed? What does that tell you about the connection’s ability to send and receive data?

- The value displays to 1 which now means the socket is still open. This means that the socket will not be able to transmit data until the socket is not closed.

While the Connection Is Still Open

- Type several messages using the Send button.

4. Does `socket.readyState` change during normal message exchange? Why or why not?

- No it does not. This is most likely due to webSockets allowing bidirectional communication.

After Clicking “Close Connection”

- Check `socket.readyState` immediately after you click Close.sdfasas
5. What value do you see while the connection is shutting down? What value do you see a few seconds later, after it fully closes?
- ☐ The readystate changes from closing to closed. So from 2 -3. Closing is the state that it is shutting down. And 3 is closed connection. So the socket connection has shut.

Reflection Questions

1 . How is a WebSocket connection different from an HTTP request?

- HTTP uses unidirectional communication, meaning the client can send a request, and the server can only respond. WebSockets, however, allow bidirectional communication, meaning both the client and server can send messages to each other at any time while the connection is open. HTTP opens a connection, sends data, gets a response, and then closes, whereas WebSockets stay open after being established and remain open until either side closes them.

2. Why are WebSockets ideal for real-time applications such as chats, dashboards, or multiplayer games?

- WebSockets are ideal for real-time applications since they allow instantaneous communication between the client and server. Because the connection stays open, the server can send updates to the client without reopening the connection each time a message is sent.

3. Why is WebSocket communication considered *stateful*?

- WebSockets are stateful because once a connection is established, the client and server maintain a continuous, shared context throughout the session. The connection remembers who the user is, what session they belong to, and can handle messages in order without re-establishing communication each time.

4. What do the `readyState` values reveal about the lifecycle of a WebSocket?

- Ready state is useful in debugging, as it helps developers manage how their code responds at each phase.
 - 0 – *CONNECTING*: The socket is being created but the connection is not yet

open.

- 1 – OPEN: The connection is established; messages can be sent and received.
- 2 – CLOSING: The connection is in the process of shutting down.
- 3 – CLOSED: The connection has been closed or could not be opened.

5. If two users connect to the same WebSocket server at the same time, how might each user's *readyState* differ during setup or closing?

- Even if both users connect at nearly the same time, their ***readyState*** values might differ slightly due to network delays, browser performance, or server response times.