WEB DEVELOPMENT

WebSockets & Real-Time Communication

The WebSocket Protocol · Raw Servers & Clients

Socket.IO vs Raw WebSockets · Rooms & Presence

Scaling · Authentication & Security

Capstone: Building a Real-Time Feature

Philip Osztromok

Generated with Claude

Table of Contents

WebSockets & Real-Time Communication — 9 Chapters

⚡ What Real-Time Communication Actually Means

HTTP was built around a simple shape: the client asks, the server answers, the connection closes. That shape has no room for the server to say something first — a new chat message, a live score, a stock price tick — without the client asking again. Every "real-time" technique in this course is really a different workaround for that one limitation, each with its own cost.

Polling: Just Ask Again

The simplest workaround — repeatedly ask the server "anything new?" on a timer:

```
setInterval(async () => {
  const response = await fetch("/api/messages/latest");
  const messages = await response.json();
  renderMessages(messages);
}, 3000);  // ask every 3 seconds, whether or not anything changed
```

The Cost

Most requests come back with nothing new — wasted requests, wasted server load, and up to a full polling interval of latency before a real update is even noticed.

Where It's Still Fine

Genuinely low-urgency data — a dashboard that refreshes every 30 seconds is polling done reasonably, not a mistake.

Long Polling: Ask, Then Wait

A refinement — the client asks, but the server holds the request open until it actually has something to say (or a timeout passes), then the client immediately asks again:

```
async function longPoll() {
  const response = await fetch("/api/messages/wait-for-next");  // server holds this open
const message = await response.json();
  renderMessage(message);
  longPoll();  // immediately ask again
}
```

Lower latency than plain polling, but it's still fundamentally request/response — every message round-trips through a brand new HTTP request, headers and all.

📡 Server-Sent Events: One-Way, Kept Simple

SSE opens one long-lived HTTP connection and lets the server push events down it whenever it wants — no new request needed per message, but strictly server-to-client only:

```
const events = new EventSource("/api/messages/stream");

events.onmessage = (event) => {
  const message = JSON.parse(event.data);
  renderMessage(message);
};
// The browser automatically reconnects if the connection drops —
// a real advantage SSE has over hand-rolled reconnection logic.
```

Genuinely Simpler Than WebSockets

Plain HTTP under the hood, automatic reconnection built into the browser, works through most proxies without special configuration.

The Limitation

One direction only — fine for a live score feed or a notification stream, useless the moment the client needs to send something back over the same channel.

WebSockets: Genuinely Two-Way

A WebSocket is a single connection that stays open and lets *either side* send a message at any time — the only one of these four techniques that's truly full-duplex:

```
const socket = new WebSocket("wss://example.com/chat");

socket.onmessage = (event) => renderMessage(JSON.parse(event.data));

sendButton.addEventListener("click", () => {
  socket.send(JSON.stringify({ text: input.value }));  // client -> server, any time
});
```

📊 Choosing the Right Tool

Technique	Direction	Latency	Complexity
Polling	Client → Server (repeated)	Up to one interval	Trivial
Long Polling	Client → Server (held open)	Low	Moderate
SSE	Server → Client only	Instant	Low
WebSockets	Both directions	Instant	Highest

Matching the Tool to the Job

SSE Is Enough

Live sports scores, a stock ticker, a notification bell, a build-status dashboard — anything where only the server ever has news to share.

WebSockets Are Needed

Chat, multiplayer games, collaborative editing, anything where the client needs to send data back over the same live connection.

💻 Coding Challenges

Challenge 1: Identify the Right Technique

For each of the following, name the most appropriate technique from this chapter and justify it in one sentence: (a) a live cryptocurrency price ticker, (b) a two-player tic-tac-toe game, (c) a "new version available, refresh?" banner.

Goal: Practice matching a real-time requirement to the least complex tool that actually satisfies it.

→ Solution

Challenge 2: Implement Naive Polling, Then Improve It

Write a polling loop that checks `/api/status` every 5 seconds, then rewrite it as long polling against a (described, not implemented) `/api/status/wait` endpoint that holds the request open until the status actually changes.

Goal: Practice feeling the concrete difference in latency and request volume between the two approaches.

→ Solution

Challenge 3: Set Up an SSE Connection

Write client-side code using `EventSource` to connect to `/api/notifications/stream`, log every incoming event, and handle the connection's `onerror` event by logging a reconnection attempt message.

Goal: Practice the basic SSE client API before WebSockets are introduced in the next chapter.

→ Solution

⚠️ Gotcha: Reaching for WebSockets by Default

WebSockets are the most powerful tool in this chapter and also the most complex to run correctly in production — connection state to manage, reconnection logic to write by hand, load balancer configuration to get right (covered in Chapter 7). A live dashboard that only ever needs the server to push updates is a worse system with WebSockets than with SSE: same real-time feel, more moving parts, more that can go wrong. Reach for WebSockets specifically when the client genuinely needs to talk back over the same connection — not by default because it sounds more "real-time."

🎯 What's Next

With the landscape mapped out, the next chapter goes under the hood of the technique this course is really about: **The WebSocket Protocol** — the HTTP Upgrade handshake, full-duplex framing, and what `ws://` and `wss://` actually mean on the wire.

🔌 The WebSocket Protocol

Chapter 1 established *why* WebSockets exist. This chapter goes under the hood of *how* they actually work — starting with a detail that surprises a lot of people: a WebSocket connection begins as a completely ordinary HTTP request.

The HTTP Upgrade Handshake

The client sends a normal-looking HTTP `GET` request, but with headers asking the server to switch protocols mid-connection:

The Raw Handshake, Side by Side

```
// Client request
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13

// Server response — note the status code
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

`Sec-WebSocket-Accept` is computed by hashing the client's key together with a fixed constant — proof the server actually understood the request, not just an echo. Once the `101` response arrives, the HTTP connection is done being HTTP — the same underlying TCP connection now carries the WebSocket protocol instead.

Why Start as HTTP at All?

Because it starts as an ordinary request, a WebSocket handshake sails through the same ports (80/443) and most proxies/firewalls that already allow normal web traffic — no special network configuration needed in most environments.

One Handshake, Then Done

The Upgrade dance happens exactly once, at connection start — everything after this is the WebSocket protocol itself, not more HTTP requests.

↔️ Full-Duplex vs Half-Duplex

Even a kept-alive HTTP connection is fundamentally **half-duplex** — one side sends a complete request, then waits for a complete response, strict turn-taking. A WebSocket connection is genuinely **full-duplex**: either side can send a message at any moment, including while the other side is still sending one.

Frames, Not More HTTP Requests

After the handshake, messages travel as lightweight binary **frames** — no headers, no status lines, just a small frame header plus the payload:

Opcode	Meaning
`0x1`	Text frame (UTF-8 payload)
`0x2`	Binary frame
`0x8`	Close
`0x9`	Ping
`0xA`	Pong

Masking (Client → Server Only)

Every frame the browser sends is XOR-masked with a random key — a security measure preventing a malicious page from crafting bytes that could poison a misbehaving proxy's cache. Server-to-client frames are never masked.

Minimal Overhead

A small text message can travel in a frame with as little as 2 bytes of header — compare that to a full set of HTTP headers on every polling request from Chapter 1.

Per-Message Overhead: Polling vs an Open WebSocket

HTTP Polling

Every single message is a brand-new HTTP request: request line, a full set of headers (cookies, user-agent, etc.), a fresh TCP handshake if the connection isn't kept alive — easily hundreds of bytes of overhead per message.

Established WebSocket

The connection and its headers exist exactly once, at handshake time. Every message after that is just a frame — a few bytes of header plus the actual payload.

🔒 `ws://` vs `wss://`

The exact same relationship as `http`/`https` from the HTTPS/TLS course — `wss://` runs the WebSocket connection over TLS, encrypting everything after the handshake the same way HTTPS encrypts a normal page:

```
const insecure = new WebSocket("ws://example.com/chat");   // plaintext — never in production
const secure   = new WebSocket("wss://example.com/chat");  // TLS-encrypted — the only real choice
```

Ping/Pong: Protocol-Level Keepalive

The protocol itself includes ping and pong frames — the server (or client) periodically sends a ping, and the other side automatically replies with a pong, letting either end detect a connection that's gone dead without any application code writing that logic. This is distinct from any application-level heartbeat message you might add on top of it.

Closing Properly: The Close Handshake

A well-behaved close exchanges close frames carrying a status code and optional reason, rather than either side just abruptly dropping the TCP connection:

Code	Meaning
`1000`	Normal closure
`1001`	Going away (e.g. page navigating off)
`1006`	Abnormal closure (connection just died — no close frame was ever received)

💻 Coding Challenges

Challenge 1: Read a Handshake

Given a captured handshake request missing the `Upgrade` header, explain what the server should do in response, and why the connection would fail to establish.

Goal: Practice recognizing the exact headers required for a valid WebSocket handshake.

→ Solution

Challenge 2: Explain Full-Duplex With a Scenario

Describe a concrete scenario in a chat application where full-duplex communication produces a noticeably better user experience than a half-duplex (request/response) alternative could.

Goal: Practice connecting the abstract protocol property to a real, observable user-facing difference.

→ Solution

Challenge 3: Diagnose a Mixed-Content Block

A page served over `https://example.com` tries to open `new WebSocket("ws://example.com/chat")` and the connection is silently blocked by the browser. Explain why, and provide the fix.

Goal: Practice connecting the ws/wss distinction to a real, commonly-hit browser security restriction.

→ Solution

⚠️ Gotcha: A WebSocket Is One Extremely Long-Lived Connection

It's easy to mentally model a WebSocket as "a request that stays open" and treat it like any other HTTP request for logging, monitoring, or load-balancing purposes — it isn't. A single WebSocket connection can live for hours, carries no natural request boundary the way HTTP does, and needs to stay pinned to the same backend server for its entire lifetime (Chapter 7 covers why). Tooling built around "count requests per second" or "route each request independently" often needs real adjustment to handle a connection that's alive, and expected to stay alive, for far longer than any normal HTTP request ever would.

🎯 What's Next

With the protocol itself understood, the next chapter puts it to work: **Building a Raw WebSocket Server** — Node's `ws` library, and handling connection, message, and close events directly.

🛠️ Building a Raw WebSocket Server

Chapter 2 covered the protocol itself — the handshake, the frames, the close sequence. None of that needs to be implemented by hand: Node's `ws` library handles the protocol details and hands you a clean event-based API. This chapter builds a real server with it.

Installing `ws`

```
npm install ws
```

A Minimal Server

```
import { WebSocketServer } from "ws";

const wss = new WebSocketServer({ port: 8080 });

wss.on("connection", (socket) => {
  // "socket" here represents this one specific client's connection
  console.log("A client connected.");
});
```

One Server, Many Sockets

`wss` is the server itself; `socket` (often named `ws` in examples) is created fresh for every individual client that connects — the `"connection"` handler runs once per client, not once total.

No Manual Framing

The library already did the handshake and frame parsing from Chapter 2 before this handler ever runs — by the time you have a `socket`, the WebSocket protocol is fully established.

The `message` Event

```
wss.on("connection", (socket) => {
 socket.on("message", (data) => {
   const text = data.toString();  // data arrives as a Buffer, not a string
const parsed = JSON.parse(text);
   console.log("Received:", parsed);
 });
});
```

Every message a client sends fires this event on that client's own `socket` object — `data` is a `Buffer` by default, so it needs `.toString()` before treating it as text or parsing it as JSON.

Sending Data Back

```
socket.on("message", (data) => {
 const parsed = JSON.parse(data.toString());
 socket.send(JSON.stringify({ echo: parsed }));  // reply to just this one client
});
```

The `close` and `error` Events

```
wss.on("connection", (socket) => {
 socket.on("close", (code, reason) => {
   console.log(`Client disconnected: ${code} ${reason}`);
   // clean up anything tracked for this socket — e.g. remove it from a room
 });

 socket.on("error", (err) => {
   console.error("Socket error:", err);
 });
});
```

👥 Tracking Multiple Clients

A single `socket` only knows about itself — broadcasting to everyone means keeping your own collection of connected sockets:

```
const clients = new Set();

wss.on("connection", (socket) => {
 clients.add(socket);

 socket.on("message", (data) => {
  for (const client of clients) {
   if (client.readyState === client.OPEN) {
    client.send(data.toString());  // broadcast to everyone, including the sender
   }
  }
 });

 socket.on("close", () => clients.delete(socket));  // stop tracking a disconnected client
});
```

This is deliberately minimal — real rooms, presence tracking, and targeted broadcasting (not "everyone") are the focus of Chapter 6.

readyState

Every socket has a `readyState`: `CONNECTING`, `OPEN`, `CLOSING`, or `CLOSED`. Checking for `OPEN` before sending avoids errors from trying to write to a socket that's already on its way out.

Sharing an HTTP Port

A `WebSocketServer` can attach to an existing `http.Server` (e.g. an Express app's server) instead of listening on its own port — common in real deployments where regular HTTP and WebSocket traffic share one port.

Raw `ws` Today vs Socket.IO (Next Chapter)

What `ws` Gives You

A thin, direct wrapper around the protocol — connection/message/close events, and nothing else. Rooms, reconnection, acknowledgements: all hand-rolled.

What a Library Like Socket.IO Adds

Automatic reconnection, a room/namespace system, message acknowledgements, and fallback transports — at the cost of a heavier abstraction and its own wire protocol on top of WebSockets.

💻 Coding Challenges

Challenge 1: Build an Echo Server

Using `ws`, write a complete server that listens on port 8080 and sends every message it receives straight back to the same client that sent it (not broadcast to everyone).

Goal: Practice the basic connection/message/send flow before adding multi-client broadcasting.

→ Solution

Challenge 2: Track Connected Clients With Logging

Extend the echo server to maintain a `Set` of connected clients, logging the current client count every time a client connects or disconnects.

Goal: Practice managing per-connection state that outlives a single message.

→ Solution

Challenge 3: Defend Against a Bad readyState

Write a `broadcast(clients, message)` helper function that safely sends `message` to every socket in a `Set`, skipping any socket that isn't currently `OPEN`.

Goal: Practice the defensive `readyState` check this chapter's tip box warns about.

→ Solution

⚠️ Gotcha: Sending to a Socket That Isn't Open Yet (or Anymore)

`socket.send()` doesn't automatically wait for the connection to be ready, and it doesn't throw a helpful error if the socket has already started closing — depending on timing, a careless call can silently fail or throw an uncaught exception. This bites most often in the broadcast pattern above: by the time a loop reaches a given client, that client may have disconnected moments earlier. Always check `readyState === WebSocket.OPEN` immediately before calling `send()` on any socket you didn't just create yourself in the current handler.

🎯 What's Next

With a working raw server in hand, the next chapter steps back to compare this approach against a higher-level library: **Socket.IO vs Raw WebSockets** — reviewing the Socket.IO coverage from the Node.js and Express courses, and figuring out when raw `ws` is actually the better choice.

⚖️ Socket.IO vs Raw WebSockets

This is a review chapter, not a fresh introduction — Socket.IO itself was already covered in depth back in Express Course 2 (express2-6: rooms, `io.use()` auth middleware, acknowledgements, namespaces). What that chapter didn't have was Chapters 2 and 3 of *this* course to compare it against. Now that raw `ws` has been built by hand, the real question can be asked properly: what does Socket.IO actually add, and what does it cost?

A Quick Recap

```
// Server
io.on("connection", (socket) => {
  socket.join("room-42");                    // no hand-rolled Set needed
  socket.on("chat message", (data, ack) => {
    io.to("room-42").emit("chat message", data);  // broadcast to just this room
ack("received");                          // acknowledgement, built in
  });
});
```

Compare that `io.to("room-42").emit(...)` line to Chapter 3's manual `for (const client of clients)` loop with a `readyState` check — that's the shape of what Socket.IO is trading away complexity for.

➕ What Socket.IO Actually Adds

Automatic Reconnection

A raw `WebSocket` that drops just closes — reconnecting is entirely the application's job. Socket.IO's client reconnects automatically, with backoff, out of the box.

Fallback Transports

If a WebSocket connection genuinely can't be established (a restrictive corporate proxy, an old environment), Socket.IO transparently falls back to HTTP long-polling underneath the same API — Chapter 1's technique, used as a safety net.

Rooms & Namespaces, Built In

No hand-rolled `Set` of clients, no manual filtering — `socket.join()`/`io.to()` and `io.of("/admin")` replace exactly the kind of bookkeeping Chapter 3 did by hand.

Acknowledgements

A built-in request/response pattern over an otherwise fire-and-forget connection — pass a callback to `emit()` and the other side can call it back. Raw `ws` requires hand-rolling this with correlation IDs.

What It Costs

Its Own Wire Protocol

Socket.IO layers its own protocol (Engine.IO) on top of WebSocket frames — a Socket.IO *client* can only really talk to a Socket.IO *server*. It is not a drop-in replacement for talking to an arbitrary raw `ws` service.

More Weight, More Moving Parts

A heavier client bundle, an extra dependency to version-match between client and server, and slightly more per-message overhead from its own framing layered on top of the WebSocket frame from Chapter 2.

Side by Side

Raw `ws`

Talks plain WebSocket — interoperable with any WebSocket client in any language. No reconnection, rooms, or acknowledgements unless you write them yourself.

Socket.IO

Reconnection, fallback, rooms, and acks all included — but locked to Socket.IO on both ends, and heavier per connection.

Dimension	Raw `ws`	Socket.IO
Interoperability	Any WebSocket client, any language	Socket.IO clients only
Reconnection	Hand-rolled	Automatic
Fallback transport	None	Long-polling fallback
Rooms/namespaces	Hand-rolled	Built in
Acknowledgements	Hand-rolled (correlation IDs)	Built in
Message overhead	Lower (raw frames)	Slightly higher

🧭 Choosing Between Them

Raw `ws` wins when you need to interoperate with a non-JavaScript client or a strict WebSocket-only spec (a game engine, an IoT device, another language's WebSocket library), when minimal dependencies matter, or when you're already planning to build custom room/scaling logic anyway (Chapters 6 and 7 do exactly this with Redis pub/sub). Socket.IO wins when the clients are all browsers, rapid development matters more than raw efficiency, and rooms/acks/reconnection out of the box save real time.

💻 Coding Challenges

Challenge 1: Translate a Raw `ws` Broadcast to Socket.IO

Take Chapter 3's manual broadcast loop (iterating a `Set` of clients with a `readyState` check) and rewrite the equivalent behavior using Socket.IO's API.

Goal: Practice seeing exactly which hand-rolled logic a Socket.IO built-in replaces.

→ Solution

Challenge 2: Justify a Technology Choice

A team is building a multiplayer browser card game with in-game chat, needing rooms per game table and reconnection support for players with flaky Wi-Fi. Recommend raw `ws` or Socket.IO, and justify the choice using this chapter's comparison table.

Goal: Practice applying the decision framework to a concrete, realistic scenario.

→ Solution

Challenge 3: Explain an Interoperability Failure

A Python service using a plain WebSocket client library tries to connect directly to a Socket.IO server and the connection fails to behave correctly. Explain why, referencing what Socket.IO actually is under the hood.

Goal: Practice explaining the Engine.IO wire-protocol distinction in your own words.

→ Solution

⚠️ Gotcha: "It Uses WebSockets" Doesn't Mean "It's Interoperable"

It's tempting to assume that because Socket.IO "uses WebSockets under the hood," any WebSocket client can talk to a Socket.IO server, or a Socket.IO client can talk to any WebSocket server — neither is true. Socket.IO's handshake and message framing (via Engine.IO) sit on top of the raw WebSocket protocol from Chapter 2, and a plain `ws` client has no idea how to speak that extra layer. If a service needs to be reachable by arbitrary WebSocket clients — not just other Socket.IO instances — build it on raw `ws`, not Socket.IO.

🎯 What's Next

With server-side approaches compared, the next chapter moves to the browser: **Client-Side WebSockets** — the native `WebSocket` API, reconnection strategies, and reflecting connection state in the UI.

🌐 Client-Side WebSockets

Chapters 2 through 4 were all server-side. This chapter moves to the browser — and immediately runs into the exact gap Chapter 4 flagged: the native `WebSocket` API has no automatic reconnection. If the connection drops, it's gone, and building something better is entirely this chapter's job.

The Native API

```
const socket = new WebSocket("wss://example.com/chat");

socket.addEventListener("open", () => {
  console.log("Connected.");
});

socket.addEventListener("message", (event) => {
  const data = JSON.parse(event.data);
  renderMessage(data);
});

socket.addEventListener("close", (event) => {
  console.log(`Closed: ${event.code} ${event.reason}`);
});

socket.addEventListener("error", (err) => {
  console.error("Socket error:", err);
});
```

The same four events from Chapter 3's server-side `ws` library, mirrored on the client: `open`, `message`, `close`, `error`. Sending is the same one method too — `socket.send(...)`, guarded by the same `readyState` check from Chapter 3.

🔁 Building Reconnection By Hand

Since nothing does this automatically, a reconnecting client needs its own retry loop — and a naive one causes real problems:

Naive Retry vs Backoff + Jitter

Naive: Retry Immediately

If the server restarts and drops every connection at once, every client reconnects in the same instant — a thundering herd that can overwhelm the server right as it's coming back up.

Backoff + Jitter

Each retry waits longer than the last (backoff), and the exact delay is randomized a little (jitter) — spreading reconnection attempts out instead of all landing at once.

A Small Reconnecting Wrapper

```
class ReconnectingSocket {
 constructor(url) {
  this.url = url;
  this.attempt = 0;
  this.connect();
 }

 connect() {
  this.socket = new WebSocket(this.url);

  this.socket.addEventListener("open", () => {
   this.attempt = 0;  // reset backoff once we're actually connected
  });

  this.socket.addEventListener("close", () => {
   const delay = Math.min(1000 * 2 ** this.attempt, 30000);  // exponential, capped at 30s
const jitter = Math.random() * 0.3 * delay;
   this.attempt++;
   setTimeout(() => this.connect(), delay + jitter);
  });
 }
}
```

🟢 Reflecting Connection State in the UI

Users need to know when the connection isn't live — a small state machine drives that:

State	UI Treatment
Connecting	Neutral indicator ("Connecting...")
Connected	Normal UI, indicator hidden or green
Reconnecting	Visible warning ("Reconnecting..."), messages queued or disabled
Disconnected (gave up)	Clear error state, manual retry option

What to Do With Messages Sent While Disconnected

Either queue them to send once reconnected, or reject them immediately with clear UI feedback — silently dropping a message the user thinks was sent is the worst option.

Cleaning Up

Close the socket explicitly on page unload or when a component unmounts in a single-page app — an abandoned open socket with live listeners is a real memory/connection leak.

💻 Coding Challenges

Challenge 1: Add a Maximum Retry Count

Extend the `ReconnectingSocket` class above so that after 10 failed reconnection attempts in a row, it stops retrying and instead exposes a way for the UI to detect the connection has been permanently given up on.

Goal: Practice turning an infinite retry loop into one with a defined, observable end state.

→ Solution

Challenge 2: Queue Messages While Disconnected

Add a `send(data)` method to `ReconnectingSocket` that queues messages if the socket isn't currently `OPEN`, and flushes the queue the moment the connection re-opens.

Goal: Practice handling the "user sends something while disconnected" case discussed in this chapter.

→ Solution

Challenge 3: Explain a Duplicate-Handler Bug

A developer's reconnection code calls `this.connect()` on every retry but never removes the event listeners attached to the previous (now-closed) socket. Explain what goes wrong as reconnections accumulate, and how to fix it.

Goal: Practice reasoning about listener lifecycle across multiple socket instances.

→ Solution

⚠️ Gotcha: Stale Listeners on a New Socket

Every reconnect in the pattern above creates a brand-new `WebSocket` object — a completely fresh instance, not the old one somehow reopened. If code elsewhere holds a reference to the *old* socket (say, a variable captured before the first disconnect) and keeps calling `.send()` on it, those calls fail silently or throw, because that specific object is permanently closed — the "live" connection is a different object entirely now. Always read the current socket through the wrapper (e.g. `this.socket`) rather than caching a raw reference to one particular `WebSocket` instance.

🎯 What's Next

With a resilient client in place, the next chapter goes back to the server to build out what a real application actually needs: **Broadcasting, Rooms & Presence** — grouping connections, targeted messaging, and tracking who's currently online.

🏠 Broadcasting, Rooms & Presence

Chapter 3's broadcast sent every message to *everyone*. Real applications need targeted messaging — a chat channel, a game table, a document's collaborators — not one global blast. Chapter 4 pointed out that Socket.IO gives you rooms for free; this chapter builds the equivalent by hand on raw `ws`, which is exactly what Chapter 7's scaling approach will extend across multiple servers.

Modeling Rooms

Instead of Chapter 3's single global `Set`, a room system needs a *collection* of sets — one per room:

```
const rooms = new Map();  // roomName -> Set of sockets
function joinRoom(socket, roomName) {
  if (!rooms.has(roomName)) rooms.set(roomName, new Set());
  rooms.get(roomName).add(socket);
  socket.rooms.add(roomName);  // tracked on the socket too — see below
}

function leaveRoom(socket, roomName) {
  rooms.get(roomName)?.delete(socket);
  socket.rooms.delete(roomName);
}
```

Tracking Rooms on the Socket

A `ws` socket is just a plain object — attaching `socket.rooms = new Set()` when a client connects gives each socket its own record of which rooms it's currently in, essential for cleanup on disconnect.

Map of Sets

The `rooms` Map holds one `Set` of sockets per room name — the same `readyState`-checked broadcast pattern from Chapter 3, just scoped to one Set at a time instead of one global Set.

Broadcasting to Just One Room

```
function sendToRoom(roomName, message) {
  const members = rooms.get(roomName);
  if (!members) return;

  for (const socket of members) {
    if (socket.readyState === socket.OPEN) {
      socket.send(message);
    }
  }
}
```

Cleaning Up on Disconnect

```
socket.on("close", () => {
  for (const roomName of socket.rooms) {
    leaveRoom(socket, roomName);  // remove from every room it was in, not just one
  }
});
```

This is why tracking rooms on the socket itself (not just in the `rooms` Map) matters — without `socket.rooms`, there'd be no way to know which rooms to clean this socket out of.

👀 Presence: Who's Online

A "who's here" feature just means notifying a room when membership changes:

```
function joinRoom(socket, roomName) {
 if (!rooms.has(roomName)) rooms.set(roomName, new Set());
 rooms.get(roomName).add(socket);
 socket.rooms.add(roomName);

 sendToRoom(roomName, JSON.stringify({
  type: "presence", event: "joined", userId: socket.userId
 }));
}
```

Chapter 3's Global Broadcast vs This Chapter's Room Broadcast

> **Chapter 3**
>
> One `Set`, one audience — every connected client, no matter what they're actually doing.

> **This Chapter**
>
> A `Map` of many `Set`s — each message reaches only the clients who actually joined that specific room.

Detecting Dead Connections Faster

An ungraceful disconnect (a dropped Wi-Fi connection, not a clean close) can take a while for the OS-level TCP timeout to notice. The ping/pong frames from Chapter 2 let the server detect a truly dead socket well before that, so presence data doesn't lag behind reality.

Single-Server Limitation

This entire `rooms` Map lives in one process's memory — a client connected to a *different* server instance has no visibility into it at all. Chapter 7 covers exactly this problem.

💻 Coding Challenges

Challenge 1: Implement `getRoomMembers`

Write a function `getRoomMembers(roomName)` that returns an array of `userId` values for every socket currently in that room (skip sockets that don't have a `userId` set).

Goal: Practice reading from the room Map structure for a "who's here" UI feature.

→ Solution

Challenge 2: Broadcast a "Left" Presence Event

Extend `leaveRoom` so that, in addition to removing the socket from the room, it broadcasts a `{ type: "presence", event: "left", userId }` message to the room's remaining members.

Goal: Practice mirroring the "joined" presence pattern for the leave case.

→ Solution

Challenge 3: Find the Memory Leak

A `joinRoom`/`sendToRoom` implementation is used in production, but the `close` event handler is never wired up to call `leaveRoom`. Explain what accumulates over time and why it's a problem, even though messages still appear to broadcast correctly at first.

Goal: Practice reasoning about a bug that isn't visible until the system has been running a while.

→ Solution

⚠️ Gotcha: Forgetting Room Cleanup on Disconnect

It's easy to wire up `joinRoom` and `sendToRoom` correctly and never notice that `leaveRoom` isn't being called on disconnect — broadcasts still appear to work fine, because `sendToRoom`'s `readyState` check silently skips closed sockets. The bug is invisible in normal testing and only shows up as a slow memory leak: every room's `Set` quietly accumulates references to long-closed sockets forever, since nothing is removing them. Always wire `leaveRoom` (for every room a socket was in) into the `close` event, not just into an explicit "leave" action a user might trigger.

🎯 What's Next

Rooms currently live entirely in one server's memory — the next chapter tackles what happens when there's more than one server: **Scaling WebSockets** — sticky sessions behind a load balancer, and a Redis pub/sub adapter for broadcasting across every instance.

📈 Scaling WebSockets

Chapter 6's `rooms` Map lives entirely in one server process's memory. The moment there's more than one server instance — for capacity, or just redundancy — that structure stops being enough. This chapter is about the specific problem persistent connections create that stateless HTTP never had to solve.

Why Stateless HTTP Doesn't Have This Problem

An ordinary HTTP request carries no memory of any previous request — a load balancer can send it to *any* healthy backend server, because every server is equally able to handle it from scratch. A WebSocket connection is the opposite: once established, it's a single, long-lived, **stateful** connection to one specific server process. That client is stuck to that one instance for as long as the connection lives.

📌 Sticky Sessions

A load balancer needs to route the *same* client to the *same* backend server for the entire life of a connection — not just for the initial handshake, but for Socket.IO's long-polling fallback too, since that makes repeated HTTP requests that all need to land on the same instance to work correctly.

How It's Usually Done

Consistent hashing on client IP, or a cookie the load balancer sets and reads on every request — either way, the same client keeps landing on the same backend.

The Real Cost

Load can get lopsided if some connections are long and heavy, and restarting one server instance drops every connection stuck to it, all at once — sticky sessions trade even load distribution for connection stability.

Why Rooms Break Across Servers

If User A is connected to Server 1 and User B to Server 2, and both are meant to be in the same chat room, Server 1's local `rooms` Map (from Chapter 6) has no way to reach User B's socket — that socket object only exists in Server 2's memory. Broadcasting "locally" is no longer broadcasting to the whole room.

📡 Redis Pub/Sub as the Fix

Instead of each server trying to reach sockets it doesn't have, every server **subscribes** to a Redis channel per room. When a server needs to broadcast, it **publishes** to Redis rather than only looping its own local Map — every subscribed server (including the one that published) receives the message and forwards it to whichever of *its own* sockets are in that room:

```
// Every server process runs this once per room it has local members in
redisSubscriber.subscribe(`room:${roomName}`, (message) => {
  sendToRoom(roomName, message);  // Chapter 6's local broadcast, now fed by Redis
});

// Publishing replaces "just call sendToRoom locally"
function broadcastToRoom(roomName, message) {
  redisPublisher.publish(`room:${roomName}`, message);
  // this server's OWN subscription callback above will also fire and
// deliver to its local members — no separate local call needed
}
```

Before and After Redis

> **Chapter 6: Single Server**
>
> One `rooms` Map, one process — every member of a room is guaranteed to be reachable locally.

> **This Chapter: Multiple Servers**
>
> Each server still has its own local `rooms` Map for its own connections, bridged to every other server's Map via Redis pub/sub — a message published on one instance reaches members on all of them.

What This Buys

Horizontal scaling (add more server instances as connection count grows) and redundancy — losing one instance no longer means losing an entire room's ability to communicate.

What It Costs

Redis becomes a critical dependency every server needs, a small added latency per broadcast, and one more moving part that can itself fail or need scaling.

Socket.IO ships an official Redis adapter that implements exactly this pattern — another example of the "built in vs hand-rolled" trade-off from Chapter 4, this time applied to scaling instead of rooms themselves.

💻 Coding Challenges

Challenge 1: Explain Why Sticky Sessions Matter for Fallback

Explain specifically why Socket.IO's long-polling fallback (Chapter 4) requires sticky sessions even when the WebSocket transport itself might not strictly need them for a single request.

Goal: Practice connecting the fallback mechanism from Chapter 4 to the sticky-session requirement from this chapter.

→ Solution

Challenge 2: Wire Up Redis Pub/Sub for Presence

Rewrite Chapter 6's presence "joined"/"left" broadcasts (originally calling `sendToRoom` directly) to instead publish through Redis, so presence updates are visible to members connected to any server instance.

Goal: Practice applying the publish/subscribe bridge to code already written in an earlier chapter.

→ Solution

Challenge 3: Diagnose a Missed Broadcast

A server instance was briefly restarting when another server published a room message via Redis. Explain what happens to that specific message for clients connected to the restarting server, and why.

Goal: Practice reasoning about pub/sub's lack of message persistence.

→ Solution

⚠️ Gotcha: Redis Pub/Sub Doesn't Remember Anything

Redis pub/sub is fire-and-forget — a message published to a channel is delivered only to subscribers currently connected at that exact moment. If a server instance is restarting, briefly disconnected from Redis, or hasn't subscribed yet, any message published during that window is simply gone for it, permanently — there's no replay, no queue, no persistence to catch up on later. This is a fundamentally different guarantee than a proper message queue (with durable storage and delivery guarantees), and it's the right trade for ephemeral real-time broadcasts like chat or presence, but the wrong tool entirely for anything that needs guaranteed delivery.

🎯 What's Next

With scaling handled, the next chapter turns to a topic that's been deferred since Chapter 3: **Authentication & Security for Real-Time Connections** — authenticating at the handshake, checking origin, rate limiting, and validating every incoming message.

🔐 Authentication & Security for Real-Time Connections

Every chapter so far quietly assumed a well-behaved client. This chapter drops that assumption — the same lessons the XSS, SQL Injection, and Authentication & Session Security courses taught about HTTP apply here too, just delivered over a connection that stays open far longer than any single request.

Authenticating Once, at the Handshake

A typical HTTP API checks auth on *every* request. A WebSocket authenticates *once*, at connect time — whatever identity is established during the handshake applies for the entire life of that connection, so getting the handshake right matters more here than it would for any single HTTP endpoint.

Where to Put the Token	Trade-off
Query string (`?token=...`)	Simple, but tokens can leak into server/proxy access logs since they're part of the URL
Cookie	Sent automatically on same-origin handshakes (it's still an HTTP request) — reuses the session cookie work from the Authentication course
First message after connect	Avoids URL/cookie exposure, but means briefly accepting an unauthenticated connection before validating it

Rejecting Before the Upgrade Completes

```
const wss = new WebSocketServer({
 port: 8080,
 verifyClient(info, callback) {
  const token = new URL(info.req.url, "http://x").searchParams.get("token");
  if (!isValidToken(token)) {
   return callback(false, 401, "Unauthorized");  // rejected before the 101 response
  }
  callback(true);
 }
});
```

Rejecting inside `verifyClient` stops a bad handshake before the Chapter 2 Upgrade completes — cheaper than accepting the connection and immediately closing it.

🌍 Origin Checking

The `Origin` header is present during the handshake, since it's still an HTTP request — checking it against an allowlist stops a malicious page on a different origin from opening a WebSocket to your server using a logged-in user's browser session. This is the exact same threat the CSRF course covers, just aimed at a WebSocket handshake instead of a form submission.

```
verifyClient(info, callback) {
  const allowed = ["https://example.com"];
  if (!allowed.includes(info.origin)) {
    return callback(false, 403, "Forbidden origin");
  }
  callback(true);
}
```

⏱️ Rate Limiting Per Connection

HTTP rate limiting counts requests. A single WebSocket connection can send unlimited messages over the same open socket — limiting needs to happen *per connection*, not per request:

A Minimal Token Bucket Per Socket

```
function attachRateLimit(socket, maxPerSecond = 10) {
 let tokens = maxPerSecond;
 setInterval(() => { tokens = maxPerSecond; }, 1000);

 const original = socket.emit.bind(socket);
 socket.on("message", () => {
  if (tokens <= 0) {
   socket.close(1008, "Rate limit exceeded");  // policy violation close code
return;
  }
  tokens--;
 });
}
```

🧪 Validating Every Incoming Message

The same "never trust input" lesson from the SQL Injection and XSS courses applies directly here — `JSON.parse`'d data is not automatically safe just because it parsed successfully. Validate structure and types before using it:

```
socket.on("message", (raw) => {
  let data;
  try {
    data = JSON.parse(raw.toString());
  } catch {
    return;  // not even valid JSON — drop it
  }

  if (typeof data.text !== "string" || data.text.length > 500) {
    return;  // wrong shape — never assume the client sent what the UI intended
  }

  broadcastToRoom(data.roomName, data.text);
});
```

Trusting the Client vs Not

Naive

Parse, then immediately broadcast whatever came through — no shape check, no auth check beyond the initial handshake, no rate limit.

This Chapter

Authenticated at handshake, origin-checked, rate-limited per connection, and every message validated before it's ever broadcast to anyone else.

💻 Coding Challenges

Challenge 1: Add Origin Checking to `verifyClient`

Combine the token-check and origin-check examples from this chapter into a single `verifyClient` function that rejects a connection failing either check, with the correct distinct status code for each failure.

Goal: Practice composing multiple handshake-time checks correctly.

→ Solution

Challenge 2: Validate a Chat Message Shape

Write a `validateChatMessage(data)` function that returns `true` only if `data` has a `roomName` (non-empty string), a `text` (non-empty string, max 500 characters), and no other unexpected fields.

Goal: Practice writing a strict shape check rather than trusting a partially-correct message.

→ Solution

Challenge 3: Identify a Stored XSS Vector

A chat client renders every incoming message with `messageDiv.innerHTML = data.text`. Explain exactly how this becomes a stored XSS vulnerability, referencing the XSS course, and provide the fix.

Goal: Practice recognizing that a WebSocket message is exactly as untrusted as any other user input.

→ Solution

⚠️ Gotcha: Rendering a Broadcast Message With `innerHTML`

It's easy to forget, in the middle of building rooms and presence and reconnection logic, that a chat message is still just untrusted user input — exactly the kind the XSS course spent an entire chapter on. If one client sends `<img src=x onerror="steal_cookies()">` as their chat text and every other client's UI renders it with `innerHTML`, that script now runs in every other user's browser the moment the room broadcasts it — a textbook stored XSS attack, just delivered over a WebSocket instead of a database-backed page load. The fix is the same one that course taught: use `textContent` instead of `innerHTML` for plain text, or run untrusted HTML through a sanitizer (like DOMPurify) if formatting genuinely needs to be preserved.

🎯 What's Next

Every piece is now in place — the final chapter is a capstone: **Building a Real-Time Feature**, a live chat application pulling together rooms, authentication, reconnection, and broadcasting from every chapter in this course.

🏁 Capstone: Building a Real-Time Feature

Every previous chapter built one piece in isolation. This capstone puts them together into a single, real feature: an authenticated live chat room with presence, reconnection, and abuse protection — nothing new is introduced here, only assembly.

What's Being Built

- A user connects with a token and joins a named room
- Everyone in the room sees who's present, and is notified when someone joins or leaves
- Messages are validated, rate-limited, and safely rendered
- A dropped connection reconnects automatically and rejoins the room it was in

🖥️ Server: Everything Combined

```
import { WebSocketServer } from "ws";

const rooms = new Map();  // Ch.6 — roomName -> Set of sockets
const wss = new WebSocketServer({
  port: 8080,
  verifyClient(info, callback) {  // Ch.8 — auth + origin, before the handshake completes
if (!["https://example.com"].includes(info.origin)) {
     return callback(false, 403, "Forbidden origin");
    }
    const token = new URL(info.req.url, "http://x").searchParams.get("token");
    const userId = verifyToken(token);  // null if invalid
if (!userId) return callback(false, 401, "Unauthorized");
    info.req.userId = userId;  // carried forward to the "connection" event
callback(true);
  }
});

wss.on("connection", (socket, req) => {
  socket.userId = req.userId;
  socket.rooms = new Set();  // Ch.6 — tracked for cleanup on close
attachRateLimit(socket);      // Ch.8 — token bucket per connection

  socket.on("message", (raw) => {
    let data;
    try { data = JSON.parse(raw.toString()); } catch { return; }
    if (!validateChatMessage(data)) return;  // Ch.8 — shape check
if (data.type === "join") joinRoom(socket, data.roomName);
    if (data.type === "chat")  sendToRoom(data.roomName, JSON.stringify({
      type: "chat", userId: socket.userId, text: data.text
    }));
  });

  socket.on("close", () => {
    for (const roomName of socket.rooms) leaveRoom(socket, roomName);  // Ch.6 cleanup
  });
});
```

`joinRoom`, `leaveRoom`, `sendToRoom`, and `attachRateLimit` are exactly the functions built in Chapters 6 and 8 — nothing here is new, only wired together.

🌐 Client: Reconnecting and Rendering Safely

```
const chat = new ReconnectingSocket(`wss://example.com/chat?token=${myToken}`); // Ch.5

chat.socket.addEventListener("open", () => {
  chat.send(JSON.stringify({ type: "join", roomName: "lobby" })); // rejoin on every reconnect too
});

chat.socket.addEventListener("message", (event) => {
  const data = JSON.parse(event.data);
  const messageDiv = document.createElement("div");
  messageDiv.textContent = `${data.userId}: ${data.text}`; // Ch.8 — never innerHTML
  chatWindow.appendChild(messageDiv);
});
```

Rejoining After Reconnect

Because `ReconnectingSocket` creates a brand-new socket on every reconnect (Chapter 5), the "join" message is sent again in the `open` handler every time — the server has no memory of a previous connection that's now gone.

Where Chapter 7 Would Slot In

This version runs on one server. Adding a second instance for scale means routing `sendToRoom` through Redis pub/sub instead of calling it directly — the room logic itself doesn't change at all.

An Honest Look Back at Chapter 4

What Was Hand-Built

Reconnection, rooms, presence, rate limiting, and message validation — every one of these exists for free in Socket.IO.

What Was Gained

Full control over every piece, no Engine.IO framing overhead, and a server reachable by any WebSocket client — not just other Socket.IO instances.

💻 Coding Challenges

Challenge 1: Add a "Leave Room" Message Type

Extend the server's `message` handler to support `{ type: "leave", roomName }`, calling `leaveRoom` explicitly rather than only relying on the `close` event's cleanup.

Goal: Practice adding a new message type to an existing validated message-handling structure.

→ Solution

Challenge 2: Show a Reconnecting Indicator

Using the connection-state ideas from Chapter 5, update the client so that the chat window displays a visible "Reconnecting..." banner between the `close` and next successful `open` event, and hides it once reconnected.

Goal: Practice reflecting connection state in the UI for this specific feature.

→ Solution

Challenge 3: Trace a Full Connection Lifecycle

Narrate, step by step, everything that happens from the moment a browser tab calls `new ReconnectingSocket(...)` to the moment a "chat" message it sends is rendered in another user's browser — naming which chapter's mechanism handles each step.

Goal: Practice holding the entire course's material together as one coherent system.

→ Solution

⚠️ Gotcha: Testing This Requires More Than One Client

A single-request HTTP endpoint can be tested in isolation — send a request, check the response. A real-time feature like this one is only actually exercised by *multiple concurrent connections* interacting: presence only means something with two clients, broadcasting only proves anything when a second client actually receives what a first one sent, and reconnection logic only matters when a connection is deliberately killed mid-session. Testing this course's capstone properly means opening multiple simulated clients against the same server and asserting on cross-client behavior — a fundamentally different testing shape than the single-request tests most of the earlier framework courses relied on.

🎓 Course Complete

This closes out **WebSockets & Real-Time Communication** — from the basic question of why HTTP can't push, through the protocol itself, a hand-built server and client, a fair comparison against Socket.IO, rooms and presence, scaling across multiple servers, security, and finally this capstone tying every piece into one working feature.