Frontend Integration Guide
Socket Contract (Client ↔ Backend)

Applies to: realtime game rooms using Socket.IO
Date: January 6, 2026

This document describes the events and payloads the frontend should send to and receive from the backend over Socket.IO. It is written to be copy-pastable for the frontend engineer implementing the client.

1. Quick start

Use the event names exactly as defined in *protocol/events* (constants). The examples below show literal strings for clarity, but your client should import the same constants (or mirror them) to avoid typos.

```
// npm i socket.io-client
import { io } from "socket.io-client";

const socket = io(BACKEND_URL, {
  transports: ["websocket"],   // preferred for games
  autoConnect: true,
});

// Join a room
socket.emit("C2S_JOIN", { room: "room-123", name: "Hussein" });

// Listen for lobby / game state
socket.on("S2C_LOBBY", (state) => {
  console.log("Lobby state:", state);
});
```

2. Connection & reconnection behavior

The server identifies players by **socket.id**. When a connection drops, the server may prune disconnected players and broadcast an updated lobby state. On reconnect, the client should re-join the room by sending **C2S_JOIN** again.

Recommended client behavior:

```
socket.on("connect", () => {
  // Always re-join after (re)connect
  socket.emit("C2S_JOIN", { room, name });
});

socket.on("disconnect", (reason) => {
  console.warn("Disconnected:", reason);
});
```

3. Event contract

The table below lists the expected events. If your backend uses different literal names (because of constants), map them 1:1. Payload fields are JSON.

Direction	Event	When to use	Payload (JSON)	Notes / Response
Client → Server	**C2S_JOIN**	Join or re-join a room	`{ room: string, name: string }`	Server replies by broadcasting S2C_LOBBY to the room.
Server → Client	**S2C_LOBBY**	Lobby snapshot after joins/leaves	`LobbyState object`	Contains players list and any lobby metadata. Treat as authoritative.
Client → Server	**C2S_INPUT**	Send player input (movement/actions)	`{ seq: number, t: number, input: {...} }`	Backend rate-limits inputs (~50ms cooldown). seq is a client incrementing id; t is client timestamp (ms).
Server → Client	**S2C_TICK**	Game tick/state update	`GameState object`	Broadcast every GAME_TICK_MS (e.g., 500ms). Update UI from this state.
Client → Server	**C2S_DROP**	Voluntary leave / drop	`{ room: string }`	Use when user clicks Leave or quits match. Server will broadcast S2C_LOBBY or S2C_GAME_OVER depending on rules.
Server → Client	**S2C_ERROR**	Request failed	`{ message: string, code?: string }`	Displayed when join fails, invalid payload, etc.

4. Payload shapes (suggested)

These are suggested shapes to keep frontend code consistent. Your backend objects may have additional fields. Always allow extra fields and prefer reading only what you need.

```
type LobbyState = {
  room: string;
  players: Array<{
    id: string;      // socket.id
    name: string;
    connected: boolean;
  }>;
  status: "lobby" | "in_game" | "ended";
};

type GameState = {
  room: string;
  tick: number;
  players: Array<{
    id: string;
    name: string;
    x: number;
    y: number;
    // ...other gameplay fields
  }>;
  // ...room-level fields (scores, timers, etc.)
};

type ClientInput = {
  seq: number; // increments by 1 per input message
  t: number;   // Date.now()
  input: {
    up?: boolean;
    down?: boolean;
    left?: boolean;
    right?: boolean;
    action?: string;
  };
};
```

5. Practical client patterns

```
// 1) Keep latest authoritative state
let lobbyState = null;
let gameState = null;

socket.on("S2C_LOBBY", (s) => { lobbyState = s; renderLobby(s); });
socket.on("S2C_TICK",  (s) => { gameState  = s; renderGame(s); });

// 2) Send inputs with sequencing
let seq = 0;
function sendInput(input) {
  socket.emit("C2S_INPUT", { seq: ++seq, t: Date.now(), input });
}

// 3) Drop/leave
function leaveRoom() {
  socket.emit("C2S_DROP", { room });
  socket.disconnect(); // optional
}
```

6. Notes & gotchas

- Inputs are rate-limited server-side (cooldown ~50ms). If you spam faster, some messages may be ignored; consider client-side throttling.
- The server broadcasts lobby updates to everyone in the room after joins/leaves; do not locally 'predict' the lobby list.
- Treat S2C_TICK / game state messages as authoritative. If you do client prediction, always reconcile to the latest server state.
- If you see S2C_ERROR, surface message to the user and log payload for debugging.