WebSocket

REST API vs Socket.IO

REST API

Request–Response model

- Each interaction requires a **new HTTP request**
- Communication is **one-way** (client → server)
- Server does **not push** updates automatically

```
Client → [HTTP Request] → Server
Server → [HTTP Response] → Client
```

Example:

```
GET /api/posts → returns all posts
```

Socket.IO (WebSocket)

Persistent connection using WebSocket

- Communication is **two-way** (client ↔️ server)
- Supports **real-time updates** and **low latency**
- Both sides can send/receive **custom events**

```
Client ⇄ [Open Socket Channel] ⇄ Server
(event-based: “message”, “typing”, “join”, …)
```

Example:

```
socket.emit("chat", "Hello!");
socket.on("chat", msg => console.log(msg));
```

We don't need JSON string anymore with WebSocket

Socket.IO fully supports all JSON objects such as dictionaries, arrays (lists), and buffers just like objects.

- It automatically serializes and deserializes them for you.
- No need for JSON.stringify or JSON.parse — Socket.IO handles that automatically.

Type	Works directly?	Example
Object `{}`	✅	`{ sessionId: "abc" }`
Array `[]`	✅	`[1, 2, 3]` or `[{...}, {...}]`
String / Number	✅	`"hello"`, `42`
Binary (Buffer/File)	✅	`socket.emit("file", buffer)`

In HTTP:

- **Request → Response** is a strict pair
- Client always sends request
- Server always returns response

But in WebSockets:

- Both sides can **send** (`emit`)
- Both sides can **receive** (`on`)
- There is no fixed request–response direction

Client -> Server Example:

Client:

```
socket.emit("update-list", [1, 2, 3, 4, 5]);
```

Server:

```
socket.on("update-list", (data) => {
  console.log(data); // [1, 2, 3, 4, 5]
});
```

Server -> Client Example:

Server:

```
socket.emit("user-list", [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" },
]);
```

Client:

```
socket.on("user-list", (users) => {
  console.log(users[0].name); // "Alice"
});
```

The .emit() function always has two parts:

1. Event name – "echo-response"

- This is like a “channel” or “label” for the message.
- The client listens for this same event name to handle e incoming data.

2. Payload (data) – message

- This is the actual content being sent, e.g., a string, object, or array.

src/socketServer.js

```
const { Server } = require("socket.io");
```

1. Imports Socket.IO

- Server comes from the socket.io library.
- It enables eal-time, bidirectional communication between clients (like browsers) and servers.

```
const io = new Server(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"],
  },
});
```

2. Creates a Socket.IO Server

- `new Server(server, {...})` attaches the Socket.IO server to an existing HTTP/HTTPS server.
- The CORS configuration (origin: "*") allows any client to connect.

```
io.on("connection", (socket) => {
  console.log(`user connected ${socket.id}`);
});
```

3. Listens for Client Connections

- When a client connects, the server logs the new onnection's socket.id.
- From here, you can also handle events like chat essages, notifications, typing indicators, etc.

Adding APIs to WebSocket

We can add more APIs:

```
io.on("connection", (socket) => {
  console.log(`User connected: ${socket.id}`);

  // Simple API 1: Echo message back to sender
  socket.on("echo", (message) => {
    ...
    socket.emit("echo-response", message);
  });
  // Other APIs
  socket.on("disconnect", () => {
    console.log(`User disconnected: ${socket.id}`);
  });
});
```

Main Server Program: index.js

Using `Express`

Using Express together with WebSockets is very common and convenient.

```
const server = http.createServer(app);
```

http.createServer(app):

- Express (app) runs on top of Node's HTTP server.
- This HTTP server is what actually listens on a port.

```
const io = new Server(server);
```

Socket.IO attaches to that HTTP server, so the same port handles both:

- HTTP requests (GET /, POST /api/...)
- WebSocket connections (real-time communication)

Why Combine Express + Socket.IO?

- One server handles everything: REST + WebSocket on the same port
- Simple setup: use Express middleware (cors, body-parser, etc.)
- Serve static files (React, HTML) and still support real-time updates
- Common pattern in production apps: Express routes + WebSocket events

```
const express = require("express");
const http = require("http"); // we need http
const cors = require("cors"); // we need cors
const socketServer = require("./src/socketServer");

const app = express();

const server = http.createServer(app);
socketServer.registerSocketServer(server);

app.use(cors());

app.get("/", (req, res) => {
  res.send("Hello server is working");
});

const PORT = process.env.PORT || 5000;

server.listen(PORT, () => {
  console.log(`App started listening at port ${PORT}`);
});
```

Without Using `Express`

```
const http = require("http");
const { Server } = require("socket.io");

const server = http.createServer((req, res) => {
  res.writeHead(200);
  res.end("Hello from plain Node.js server!");
});

const io = new Server(server);
io.on("connection", (socket) => {
  console.log("Client connected:", socket.id);
});

server.listen(5000);
```

- Works perfectly fine — WebSocket doesn't need Express.
- But you lose easy routing, middleware, and JSON parsing.

Client Side Programming

We can use any programming langauge to connect to servers using WebSocket.

Node.js Example

```
const io = require('socket.io-client');

// Connect to server
const socket = io('http://localhost:5000');

console.log('Connecting to Socket.IO server...\n');
```

We use `socket.io-client` package and use the same port to start communication.

Emit messages (Make Requests or Send an Event)

```
socket.on("connect", () => {
  console.log("✓ Connected to server");
  console.log(`✓ Socket ID: ${socket.id}\n`);

  // Emit various test events
  socket.emit("echo", "Hello, server!");
  socket.emit("broadcast", "Hi everyone!");
  socket.emit("join-room", "test-room");
  socket.emit("room-message", { room: "test-room", message: "Hello room!" });
  socket.emit("get-time");

  console.log("✓ All test events emitted\n");
});
```

Listen for server responses (Receive Responses)

```
socket.on("echo-response", (message) => {
  console.log(`[Echo Response] ${message}`);
});

socket.on("broadcast-message", (data) => {
  console.log(`[Broadcast] From ${data.from}: ${data.message}`);
});

socket.on("room-joined", (roomName) => {
  console.log(`[Room] Joined: ${roomName}`);
});

socket.on("room-message-received", (data) => {
  console.log(`[Room Message] From ${data.from}: ${data.message}`);
});

socket.on("current-time", (time) => {
  console.log(`[Server Time] ${time}\n`);
});
```

Disconnect

Disconnect after 3 seconds to allow responses to arrive

```
socket.on("disconnect", () => {
  console.log("✗ Disconnected from server");
});

setTimeout(() => {
  console.log("Disconnecting...");
  socket.disconnect();
  process.exit(0);
}, 3000);
```