Integrating Express with Socket.IO

1. Basic HTTP Server

In the beginning, we used Node.js's built-in http module to create servers directly:

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end("Hello World");
});

server.listen(3000);
```

- Here, requests are handled directly using the http module.
- It is low-level and requires manual handling of routing, headers, etc.

2. Using Express

Express is built on top of the http module and simplifies routing and middleware handling:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send("Hello from Express");
});

app.listen(3000);
```

- Express internally uses http.createServer(app).
- app.listen() is just a wrapper around http.createServer().

3. Adding Socket.IO

Sockets require direct access to the underlying HTTP server to handle persistent, bidirectional connections. That's why we explicitly create the server with http.createServer(app) and pass it to Socket.IO:

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

const app = express();
```

```
const server = http.createServer(app);
const io = new Server(server);

app.get('/', (req, res) => {
  res.send("Hello from Express + Socket.IO");
});

io.on('connection', (socket) => {
  console.log('A user connected');

  socket.on('chat message', (msg) => {
    console.log("Message: " + msg);
    io.emit('chat message', msg);
  });
});

server.listen(3000, () => {
  console.log('Listening on *:3000');
});
```

4. Why Both HTTP and Express?

1. Express (app)
   - Handles normal HTTP routes (GET, POST, etc.).
   - Useful for serving HTML, APIs, and static files.

2. HTTP (server)
   - Required by Socket.IO to manage WebSocket handshakes.
   - app.listen() hides the HTTP server, so we explicitly create it.

5. Flow Diagram

Client Request → HTTP Server → Express (for routes)
↘ Socket.IO (for WebSockets)