Real time data streaming with WebSocket and Socket.IO
Lab 2
Socket.IO features and usage

0 REFERENCES 2

1 BASIC WEBSOCKET / SOCKET.IO PROJECT STRUCTURE 2

1.1 KEY POINTS ABOUT CODE 5
1.1.1 server.js 5
1.1.2 client.js 5

2 BASIC SOCKET.IO APP 6

2.1 KEY POINTS ABOUT CODE 8
2.1.1 server.js 8
2.1.2 client.js 9
2.1.3 index.html 11
2.2 SUMMARY OF DIFFERENCES (WEBSOCKET VS SOCKET.IO) 11

3 BROADCASTING 11

3.1 KEY POINTS ABOUT CODE 12
3.1.1 server.js 12

4 NAMESPACES 14

4.1 KEY POINTS ABOUT CODE 15
4.1.1 server.js 15
4.1.2 client.js 16
4.2 SUMMARY OF NAMESPACES 17

5 NAMESPACES WITH ROOMS 17

5.1 KEY POINTS ABOUT CODE 18
5.1.1 server.js 18
5.1.2 client.js 20

6 SPECIFYING AND DISPLAYING USERNAMES 20

6.1 KEY POINTS ABOUT CODE 21
6.1.1 server.js 21
6.1.2 client.js 22

7 ACCESSING ROOM INFO AFTER AUTO RECONNECTION 22

7.1 KEY POINTS ABOUT CODE 24
7.1.1 client.js 24

8 SIMULATION OF REAL TIME DATA STREAMING OF MACHINE METRICS 25

8.1 KEY POINTS ABOUT CODE 26
8.1.1 server.js 26
8.1.2 index.html 26
8.1.3 styles.css 27

8.1.4 client.js 27
9 EXTENDING VISUALIZATION TO INCORPORATE NAMESPACES WITH ROOMS 27
9.1 KEY POINTS ABOUT CODE 28
9.1.1 server.js 28
9.1.2 client.js 29
10 END 30

0 References

Official documentation and tutorial:
https://socket.io/docs/v4/tutorial/introduction

https://www.w3schools.com/nodejs/nodejs_socketio.asp

1 Basic WebSocket / Socket.IO project structure

Creating a WebSocket or Socket.IO project is very similar to that of creating an Express.JS app.

Create a directory `firstapp` and open a DOS prompt in it. Type:

```
npm init
```

Press Enter to accept the default for all the questions except for the one below, for which you should enter a new value:

```
entry point (index.js): server.js
```

Press Enter to the final `Is this Ok` question and you will be returned to the prompt.

Open this folder in Visual Studio Code (or the particular IDE you are using).

You should now see a single file `package.json` in this folder. This was created by `npm init`, which is often the first command you run when starting an Express.js or any Node.js project.
The `package.json` file is the main configuration file for any Node.js project (which can be Express.js or any other related JavaScript library / framework). It serves as the project manifest. Its purpose is to:

a) Defines basic project metadata (name, version, description, author, license, etc.).
b) Lists the dependencies and devDependencies that your project requires.
c) Specifies scripts (like `npm start`, `npm test`) that automate tasks.
d) Provides information to package managers (`npm, yarn, pnpm`) for installing the project (when the command `npm install` is run)
e) Helps ensure portability and reproducibility — another developer can just run `npm install` and set up the environment.

The key fields within it are:

- **`name`** / **`version`** → Identifies the project.
- **`description`** → Optional, useful for documentation or publishing.
- **`main`** → Entry point file (e.g., `index.js` or `server.js`).
- **`scripts`** → Custom command shortcuts (`npm run <script>`).
- **`dependencies`** → Packages needed for runtime (Express, WS, etc.).
- **`devDependencies`** → Packages needed only for development (Nodemon, testing tools, etc.).
- **`keywords`** / **`author`** / **`license`** → Metadata, helpful if publishing to npm.

Now, we will install the Websocket library in this directory by typing:

```
npm install ws
```

These will download the required packages for the WS library into a new subfolder called `node_modules` and also create an additional file called `package-lock.json`

The `node_modules` folder basically contains all the package dependencies listed in `package.json`. For e.g. `package.json` currently specifies a specific version of Express as its dependency. However, the Express framework itself consists of multiple packages, which in turn can depend on other packages (nested / transitive dependencies). All these package dependencies are stored as separate folders within `node_modules` folder

The source code files for this lab can be obtained from `labcode / Lab 2` folder from the downloaded zip.

Create a file `server.js` into this project folder `firstapp` (you had earlier specified this file as the main entry point for this project in `package.json`).

Replace it with the content from `labcode` folder
`server-v1.js`

Start up the back end app which implements a simple WebSocket server functionality from the command prompt with:

```
node server.js
```

Create two additional files in `firstapp`
`index.html`
`client.js`

Replace it with the content from `labcode / public` folder
`index-v1.html`
`client-v1.js`

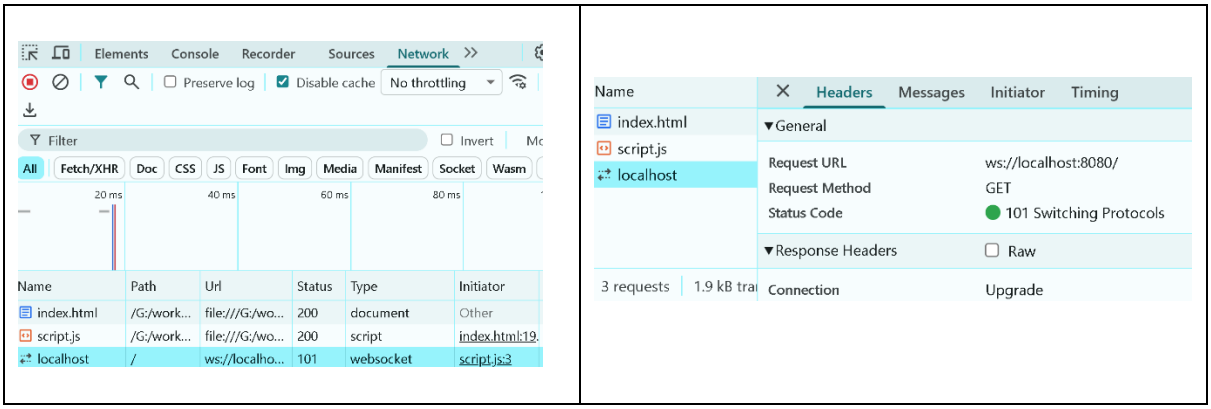
If you are using VS Code, it would be easier if you can view these source code files in two separate editor tabs (Split Right) to enable easier study and comparison of the relevant related code snippets. You can also enable Word Wrap and minimize the Explorer Pane to provide more space to view the source code.

Finally, open `index.html` in a browser tab, which will also access `client.js`

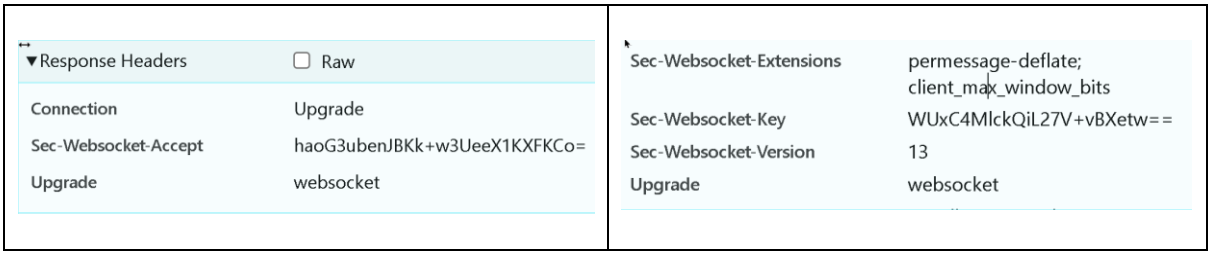
Type in various messages into the text box in the web page and click Send to send them to the backend app via the established WebSocket connection. Check that the messages are received correctly on the server side (via the console log output messages) and also sent back to the client side to be displayed again. Finally, typing Exit will terminate the connection.

You can check the network interaction between the front end app (`script.js`) and the backend app using WebSocket via the Network tab in Developer Tools view of your particular browser.

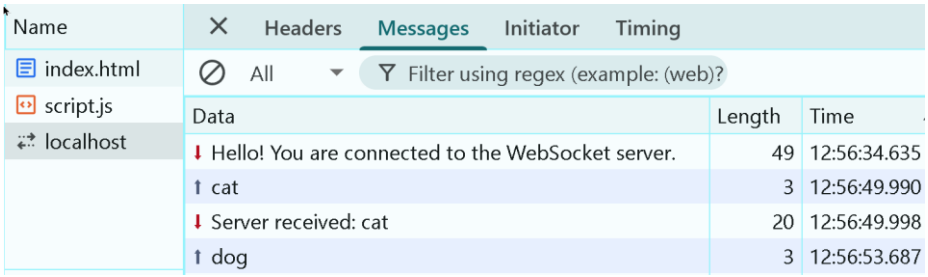
Reload the HTML to see the sequence of network interactions from the start. Notice the initial GET request from the front end with a response status code of 101 indicating success in upgrading / switching from HTTP to WebSocket.



Note the relevant WS-related headers in both the request and response: `Sec-WebSocket-Accept`, `Upgrade`, `Sec-websocket-extensions`, `Sec-websocket-key`, `Sec-websocket-version` etc



Notice that as you interact with the webpage to send messages from the front end to the back end, the exchange of messages between both client and server **occurs over the single WebSocket connection without any further HTTP GET method calls**. You can check out the data exchanged via the messages tab in Network tab, which also provides info on their length and the time the call was made.



Continue typing random messages and then type Exit to terminate the connection.
After that, if you attempt to further send any messages from the front end, you will get a message indicating that the WS connection has being closed.

Press Ctrl+C to terminate the back end app.

1.1 Key points about code

1.1.1 server.js

- `require('ws')` imports the WebSocket library used on the Node.js side.
- `new WebSocket.Server()` creates a WebSocket server instance that listens for incoming client connections.

- The event `'connection'` is fired each time a client establishes a WebSocket handshake.
- The `ws` parameter represents that individual WebSocket connection.
- `ws.send()` is used to send a message from server → client once the connection is open.

- The `'message'` event is triggered whenever the connected client sends a message.
- The message is first converted to a string and processed.
- The server then echoes a response back using `ws.send()`.

- If the client sends "exit," the connection is closed using `ws.close()`.
- The `'close'` event fires when either side terminates the WebSocket connection (due to user exit, network loss, or manual closure).

1.1.2 client.js

```
const socket = new WebSocket(`ws://localhost:${portToUse}`);
```

- Uses the browser's native WebSocket API to open a connection to the specified address.
- The scheme ws:// indicates that this is a WebSocket connection (not HTTP).

```
socket.addEventListener('open', () => { ... });
socket.addEventListener('message', (event) => { ... });
socket.addEventListener('close', () => { ... });
```

```
socket.addEventListener('error', (error) => { ... });
```

These event listeners are core WebSocket event hooks:

- `'open'`: fired when the connection is successfully established.
- `'message'`: triggered when the server sends data.
- `'close'`: triggered when the connection closes.
- `'error'`: triggered when an error occurs.

Each event uses a callback to log the event details on the page via the `log()` function.

```
if (socket.readyState === WebSocket.OPEN) {
  socket.send(text);
}
```

- The `send()` method transmits a message from client → server.
- `readyState` ensures the connection is open before sending.

2 Basic Socket.IO app

We will extend the simple back end app (`server.js`) so that it functions as well as a web server using the Express.js library, and serves back the index.html and client.js via a specified route (root route /). We also refactor the back end app now to use the Socket.IO library to establish a real-time Socket.IO connection between client and server and performing the same functionality as previously:

- Echoes all messages received from the client (`client.js`)
- Closes the connection when the client sends "Exit".

Create a directory `secondapp` and open a DOS prompt in it. Type:

```
npm init
```

Press Enter to accept the default for all the questions except for the one below, for which you should enter a new value:

```
entry point (index.js): server.js
```

Press Enter to the final `Is this Ok` question and you will be returned to the prompt.

Open this folder in Visual Studio Code (or the particular IDE you are using).

In the command prompt in the current project folder, install the Express and Socket.io libraries with:

```
npm install express socket.io
```

Create a file `server.js` into this project folder `secondapp`

Replace it with the content from `labcode` folder
`server-v2.js`

Create a subfolder `public` in `secondapp` and create these two files in there:
`index.html`
`client.js`

Replace it with the content from `labcode / public` folder
`index-v2.html`
`client-v2.js`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

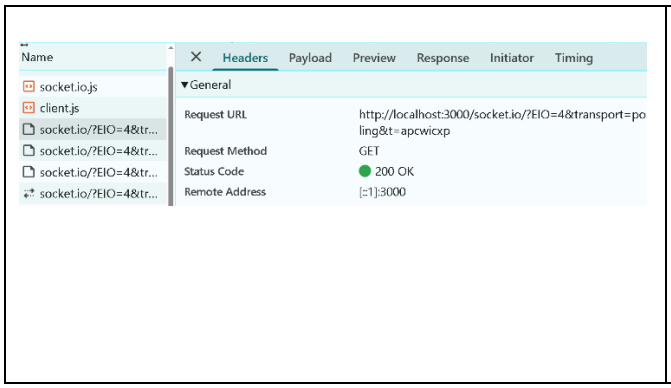
http://localhost:3000/

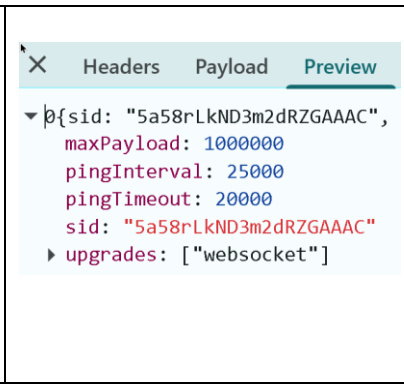
Interact with the app as we have done previously before.

Type in various messages into the text box in the web page and click Send to send them to the backend app via the established Socket.IO connection. Check that the messages are received correctly on the server side (via the console log output messages) and also sent back to the client side to be displayed again. Finally, typing Exit will terminate the connection.

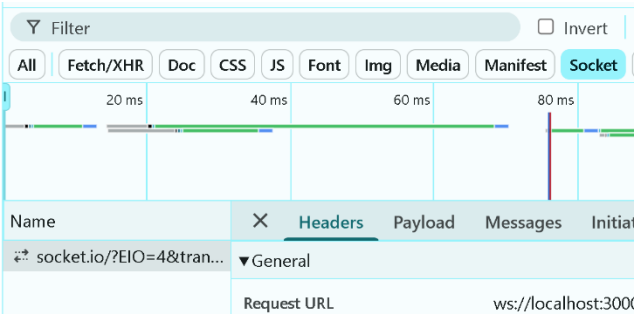
You can check the network interaction between the front end app (`client.js`) and the backend app using Socket.IO via the Network tab in Developer Tools view of your particular browser.

Reload the app to see the sequence of network interactions from the start.
Notice that the initial protocol exchanges indicates that from the client side, Socket.IO starts off with HTTP long polling as its low-level transport, before eventually requesting a switch to WebSocket (the preferred low-level transport). The reason for this is that HTTP long polling is guaranteed to work with all networks, while some internal networks block WebSocket.

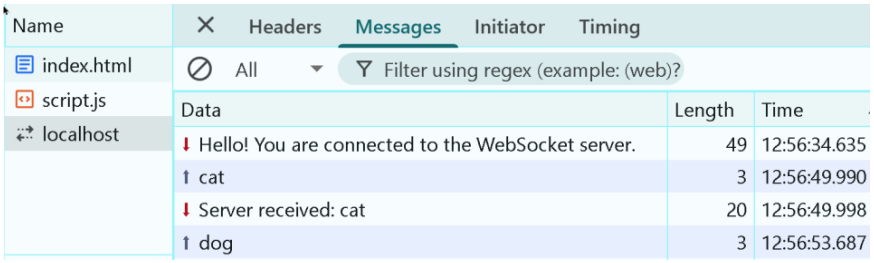




You can filter the existing network interactions in Network view to see only the Socket connections (WebSocket / Socket.IO).



Notice that as you interact with the webpage to send messages from the front end to the back end, the exchange of messages between both client and server **occurs over the single WebSocket connection without any further HTTP GET method calls**. You can check out the data exchanged via the messages tab in Network tab, which also provides info on their length and the time the call was made.



Continue typing random messages and then type Exit to terminate the connection.
After that, if you attempt to further send any messages from the front end, you will get a message indicating that the WS connection has being closed.

Press Ctrl+C to terminate the back-end app.

2.1 Key points about code

2.1.1 server.js

`io` represents the Socket.IO server instance — the overall "hub" managing all connected clients. It sits on top of your HTTP server (server) and manages WebSocket / long-polling connections, connected clients and other features such as name spaces and rooms, broadcasting messages, etc.

`socket` represents a single connection between the server and a specific client.
It appears as a parameter in the event handler / callback handler for `io.on('connection', (socket) => {…..}).` which will run when a WebSocket connection is established with a client. If 5 clients connect, this connection callback handler will run 5 times, each time with a different `socket` object.

Later on, we will see how we can use the `socket` object to:

a) Listen for events (messages) from that specific client

b) Send messages to just that specific client
c) Broadcast to other clients that don't include this specific client
d) Handle disconnection from that client

`socket.emit(eventName, data)` is the **core mechanism** in Socket.IO for sending events (in this case messages) from one side (server or client) to the other.

- eventName → the **name of the custom event** you are sending
- data → the **payload** (any value or object) that you want to send along with it

Socket.IO communication is event-based. This means you can define any event names you want, for e.g. `eventName` could be `'message'`, `'chatMessage'`, `'userJoined'`, `'welcomeEvent'`, `'systemNotice'`

The important thing is that the sender and receiver **use the same event name**.

For e.g. on `server.js`, the sending of the `message` event

```
socket.emit('message', 'Hello! You are connected to the Socket.IO server.');
```

will be processed by the corresponding event / callback handler in `client.js`

```
socket.on('message', (msg) => {
});
```

This applies vice versa as well, for e.g. in `client.js`, the sending of the `message` event

```
  if (text) {
    socket.emit('message', text);
```

will be processed by the corresponding event / callback handler in `server.js`

```
  socket.on('message', (msg) => {
    console.log('Received from client:', msg);
```

The data payload of the method call
`socket.emit(eventName, data)`
can be any value like a

- string (in this case)
- number
- boolean
- array
- JavaScript object

If you use an object, there is no restriction on what key or structure this object must have.

The most important point is the receiving side must know how to interpret it correctly (i.e. correct type and correct key-value pairs, for the case of an object).

2.1.2 client.js

The first statement establishes a connection to the backend server on the single WebSocket / Socket.io connection.

```
const socket = io();
```

The remaining `socket.on(…….)` event handlers will listen for and process specific events.

We already saw:

```
socket.on('message', (msg) => {
  log('Received from server: ' + msg);
});
```

processes `message` events that are sent via a corresponding `emit` method in `server.js`

```
socket.emit('message', 'Hello! You are connected to the Socket.IO server.');
```

The call `socket.on(….)` can use any `eventName`, for e.g. `eventName` could be `'message', 'chatMessage', 'userJoined', 'welcomeEvent', 'systemNotice'`, as long as this corresponds to what was defined on the server-side `emit` method call.

However, there are certain **predefined lifecycle events** for the `socket` object on both client and server sides which have specific reserved names that have an explicit meaning (we call this **built-in event names**).

Event name	When it occurs	Who triggers it	Common use
`'connect'`	When a new client successfully connects to the server	Socket.IO itself	Initialization, welcome messages
`'disconnect'`	When the client connection is closed or lost	Socket.IO itself	Cleanup, notifications, resource release
`'connect_error'`	When connection fails (e.g. server unreachable)	Socket.IO itself	Debugging / retry logic
`'error'`	When a transport or server error occurs	Socket.IO itself	Error handling

In that situation, we **MUST** use the explicit event name in our code if we want the corresponding call back handler to run when that event occurs, for e.g.

```
  socket.on('disconnect', () => { ….
```

Similarly the `io` object, which represents the Socket.IO server instance, has a set of predefined built-in events with specific names that have explicit meanings:

Event name	Fired when	Typical use
`'connection'` / `'connect'`	A new client connects	Set up per-client handlers

Event name	Fired when	Typical use
`'new_namespace'`	A new namespace is created dynamically	Namespace management
`'error'`	Internal server errors occur	Logging/debugging

For e.g. the most common call for a built-in event is responding to the event of a new client successfully connecting to the Socket.IO server

```
io.on('connection', (socket) => {
…….
});
```

2.1.3 index.html

The script tag
`<script src="/socket.io/socket.io.js"></script>`
automatically serves the Socket.IO client library from the server so that is can be utilized by the JavaScript program running in the browsers. Hence, there is no additional need to install separately the way we have done on the server side with `npm install socket.io`

2.2 Summary of Differences (WebSocket vs Socket.IO)

Feature	WebSocket (ws)	Socket.IO
Protocol	Raw WebSocket	Custom layer over WebSocket + fallback to HTTP long-polling
Setup	Manual connection handling	Simplified events (`connect`, `message`, `disconnect`)
Client Library on Browser	Browser-native `WebSocket`	Custom `socket.io-client` (auto-served from server)
Reconnection	Manual	Automatic
Compatibility	WebSocket-only	Works even if browser or proxy blocks WS (falls back to HTTP long polling)

3 Broadcasting

One of the key features in Socket.IO is the ability to send an event / message to all connected clients or to a subset of clients.

We will continue to work with the existing `server.js` in this project folder `secondapp`

Replace it with the content from `labcode` folder

`server-v3.js`

We will use the existing files in the subfolder `public` in `secondapp`:
`index.html`
`client.js`

Replace it with the content from `labcode / public` folder
`index-v3.html`
`client-v3.js`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

http://localhost:3000/

Open this URL in a couple of separate tabs in the same or different web browser. Notice that each loading of the client script running in a new browser tab is registered as a separate individual client with the server app.

Type a message in each message box, and notice that the message is broadcast to all clients registered with the server app.

Closing any of the open browser tabs for a registered client automatically closes the connection, this is picked up the server app which in turn broadcasts this event to all the remaining clients.

Refreshing any of the open browser tabs reloads the client scripts, which registers as a completely new client to the server app.

Connection from each of the clients in each of the browser tab is through a standard single WebSocket connection.

Press Ctrl+C to terminate the back end app when you are done.

3.1 Key points about code

3.1.1 server.js

`socket` represents a single connection between the server and a specific client. It appears as a parameter in the event handler / callback handler for `io.on('connection', (socket) => {…..}.` which will run when a WebSocket connection is established with a client .

Within the callback handler for this connection event, we can use the `socket` object to:

a) Listen for events (messages) from that specific client:
```
socket.on('chatMessage', (msg) => { ... });
```

b) Send messages to just that client
`socket.emit('message', 'Hello just to you!');`

c) Handle disconnection from that client
`socket.on('disconnect', () => { ... });`

d) Send a message to **ALL connected** clients **except** the current client that triggered the connection event
`socket.broadcast.emit(event, data)`

When a new client connects, it is useful to send a message to all other existing connected clients notifying them that a new client joined. The new client itself does not receive this broadcast message. Only the other connected clients receive it.

Typical use case for this call is to notify other clients about an event performed by one client (e.g., "user joined", "user is typing", "user left", etc).

On the other hand
`io.emit(event, data)` - Sends a message to **ALL connected** clients, **including the one** that triggered the event (send the message)

This is because the `io` object represents the Socket.IO server instance — the overall "hub" managing all connected clients. It sits on top of your HTTP server (server) and manages all connected clients and other Socket.IO complementary features such as name spaces and rooms, broadcasting messages, etc.

Feature	`socket.broadcast.emit(...)`	`io.emit(...)`
Who receives	All connected clients **except the sender**	All connected clients **including the sender**
Use scope	Must be called within a specific `socket`'s callback handler context (`io.on('connection', socket => {...})`)	Can be called anywhere in server code (global scope)
Sender receives message?	❌ No	✅ Yes
Typical use cases	"Notify everyone else" type events (join, leave, typing indicator)	"Update everyone" type events (chat messages, broadcasted data updates)
Example usage	`socket.broadcast.emit('user-joined', user)`	`io.emit('chatMessage', message)`

To summarize
a) `socket.emit()` → send only to a single connected client
b) `socket.broadcast.emit()` → send to everyone EXCEPT the current client that triggered the connection

c) `io.emit()` → send to ALL connected clients

We saw earlier that the call `socket.emit(eventName, data)` can use any `eventName`, for e.g. `eventName` could be `'message'`, `'chatMessage'`, `'userJoined'`, `'welcomeEvent'`, `'systemNotice'`
, as long as it is matched with the same eventName in the `socket.on(eventName, eventHandler)` on the client side (`client.js`)

Most of the `socket.emit` method calls here involve sending an object, rather than a simple string or number. For e.g.

```
socket.emit('message', { sender: 'Server', text: `Welcome ${clientName}!` });
…
…
socket.broadcast.emit('message', { sender: 'Server', text: `${clientName} joined the chat.` });
…
…
```

Any data or value can be sent using `emit`, as long as the receiving side knows how to interpret it correctly. For the case of an object, this means knowing what are the key names to extract from the object properties, for e.g. in `client.js`, we know to extract the `sender` and `text` properties from the object sent from the server

```
socket.on('message', (data) => {
  appendMessage(data.sender, data.text);
});
```

4 Namespaces

One of the key features in Socket.IO is the ability to organize and isolate parts of the app logic by associating independent communication channels with different namespaces.

We will continue to work with the existing `server.js` in this project folder `secondapp`

Replace it with the content from `labcode` folder
`server-v4.js`

We will use the existing files in the subfolder `public` in `secondapp`:
`index.html`
`client.js`

Replace it with the content from `labcode / public` folder
`index-v4.html`
`client-v4.js`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

http://localhost:3000/

Open this URL in a couple of separate tabs in the same or different web browser. As before, each loading of the client script running in a new browser tab is registered as a separate individual client connection with the server app.
When connecting, a client can choose which chat namespace `(/general, /sports, /movies)` to join. Once a client is within a particular namespace, all the broadcast messages from that client will be limited to other existing connected clients within the same namespace. Other namespaces will not receive that message.
We can also keep track of the number of connected clients within each namespace.
Clients within a namespace can also disconnect themselves explicitly by clicking on the Disconnect button, and this event will also be broadcast to only clients within that namespace.

Create multiple client connections for each of these 3 namespaces in different browser tabs, and experiment with joining, sending messages and leaving the namespace to see the effects.

Connection from each of the clients in each of the browser tabs is through a standard single WebSocket connection, as before.

Press Ctrl+C to terminate the back-end app when you are done.

4.1 Key points about code

4.1.1 server.js

We use `io.of(namespaceName)` to access an existing (or create a new) unique namespace object based on its name. When we start up `server.js`, we first create 3 separate namespaces `(/general, /sports, /movies)` through 3 calls to the `setupNamespace` function

Each namespace behaves like a completely independent event hub:

a) It has its own `.on('connection')` event, which is isolated from other namespaces
b) Any broadcast messages are only to clients within this namespace (for e.g. using `nsp.emit(...)`), clients connected to other namespaces never see these messages.
c) If the namespace doesn't exist yet, `.of()` automatically creates it.

When we listen for incoming connections, we use the `namespace` object:
```
nsp.on('connection', (socket) => { ... });
```
which registers an event handler / callback handler for the connection event within this specific namespace object only, which fires every time a new client connects to that namespace only.

Contrast this with the previous version, where we listen for incoming connections on the global `io` object:
```
io.on('connection', (socket) => { ... });
```

In the expression:
`const clientName = \${namespaceName.slice(1)}_User${clientCount}`;`
`namespaceName.slice(1)` removes the first character /, giving 'sports'.

Each namespace `(/general, /sports, /movies)` keeps track of its own active users using the `nsp.sockets.size` propery, which is broadcast to all connected clients associated with a namespace object `nsp`.

```
    nsp.emit('userCountUpdate', { count: nsp.sockets.size });
```

Notice that this is different from the other API call we have seen:
`io.emit(event, data)`
which will send a message to **ALL connected** clients regardless of which namespace they are in, because the `io` object represents the Socket.IO server instance — the overall "hub" managing all connected clients.

4.1.2 client.js

AT the point when we connect, we need to specify the namespace we connect to, which we first obtain from the user via a drop down list in the UI for which the event handler is provided:

```
connectBtn.addEventListener('click', () => {
```

we then connect using that namespace i.e.:

```
socket = io(namespace);
```

Contrast this with the previous version where we just simply made a connection to the single Socket.IO connection.

```
const socket = io();
```

The `socket.on(……)` callback handlers work the same way as they have for the previous example to:

- Receive new message from the server within the specified namespace
- Receive live use count updates for this namespace
- Check whether the current client is disconnected from the chat

The `appendMessage` helper function works on the DOM to append a child element with required content to the display area.

We can then send chat messages when the chat button is clicked in the appropriate event handler:

```
  sendBtn.onclick = () => {
…
…
      socket.emit('chatMessage', message);
```

4.2 Summary of namespaces

Property	Description
Purpose	Organize and isolate parts of your app logic.
Connection	Each namespace is an independent channel. Clients must explicitly connect to it.
Event handling	Each namespace has its own set of events (e.g. `connection`, `disconnect`, custom events) and corresponding event handlers
Broadcast scope	`io.of('/chat').emit()` broadcasts to all clients in that namespace only.
Isolation	Clients in one namespace cannot directly receive events from another namespace.

5 Namespaces with rooms

The rooms feature in Socket.IO further complements the use of namespaces to further organize and isolate parts of the app logic by creating subdivisions within a given namespace, which are also isolated from each other similar to namespaces.

We will continue to work with the existing `server.js` in this project folder `secondapp`

Replace it with the content from `labcode` folder
`server-v5.js`

We will use the existing files in the subfolder `public` in `secondapp`:
`index.html`
`client.js`

Replace it with the content from `labcode / public` folder
`index-v5.html`
`client-v5.js`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

http://localhost:3000/

Open this URL in a couple of separate tabs in the same or different web browser.

All the interactions that we have performed previously can be repeated here, with the addition of additional functionality for the selection of a room (Room A, Room B, Room C) by a client after connecting to the server within a specific namespace.

Once a client is within a particular namespace / room, all the broadcast messages from that client will be limited to other existing connected clients within the same namespace / room. Other namespaces / rooms will not receive that message.

We can also keep track of the number of connected clients within each room of a namespace. Clients within a room can also disconnect themselves explicitly by clicking on the Disconnect button, and this **event will also be broadcast to only clients within a room of that namespace**.

Clients within a namespace / room can also disconnect themselves explicitly by clicking on the Disconnect button, and this event will also be broadcast to only clients within that namespace / room.

Create multiple client connections for each of these 3 namespaces in different browser tabs, and experiment with joining, sending messages and leaving the namespace / rooms to see the effects.

Notice that if you were to stop the Node server and restart it again after a while, if there are still any remaining active clients running in an open browser tab, the client will automatically reconnect after a while and messages indicating this connection will be shown on both the server log output and client UI. This demonstrates the **automatic reconnection feature of Socket.IO which occurs automatically in the background**.

Nonetheless, the reconnected clients will not be able to exchange messages because the **reconnection attempt is to the previously specified namespace of a Socket.IO connection**. You can try this out by attempting to send messages amongst the reconnected clients.

The reason for this is because in the current implementation, messages are broadcast within a specific room in a namespace, and the client needs to also explicitly rejoin a room within a namespace. This is currently not implemented in code yet, and we will see how to do this in an upcoming example.

Press Ctrl+C to terminate the back-end app when you are done.

5.1 Key points about code

5.1.1 server.js

In `setupNamespace` code functionality still starts off with `io.of(namespaceName)` to access an existing (or create a new) unique namespace object based on its name, similar as before.

Then additionally, with:
`socket.on('joinRoom', …)`
we listen for a `joinRoom` event from the client, telling the server which room inside the namespace this client wants to enter.

Key point: Each socket (corresponding to a unique client connection) is automatically considered to be in **its own private room by default**, which is **given by its socket ID**. It can also join other user-created rooms, and a single socket can be in **multiple rooms simultaneously**, which is tracked by the property **socket.rooms**.

In the `socket.rooms` iteration, we remove any previously joined chat room (`socket.leave(r)`) so that the user only stays in one chat room at a time – which is the situation for this simple app.
For other use cases, a socket can and should be part of multiple rooms at the same time so that it can receive messages from all these rooms. For e.g. rooms can be conceptually similar to chat groups in WhatsApp or Slack, or multiplayer game lobbies (room for current match, room for team, room for spectators)

Then, we perform
`socket.join(roomName)`
to add this socket to a logical subgroup (room) within the namespace.

Then
`socket.to(roomName).emit(…)`
broadcasts a message to everyone else in that room (except the sender).

We can check how many sockets are in the room
`nsp.adapter.rooms.get(roomName)?.size`
Socket.IO's "adapter" maintains all namespace-room membership maps.

Finally
`nsp.to(roomName).emit('roomCountUpdate', …)`
sends a custom event so all clients in that room can update their "room user count" display.

In `socket.on('chatMessage', (msg) => {` callback handler:
we check which room the incoming message from the current socket (corresponding to a unique client connection) is in, so that we can broadcast this message to all other clients in the same room.
We saw that each socket is automatically considered to be in **its own private room by default**, which is **given by its socket ID**. It can also join other user-created rooms, and a single socket can be in **multiple rooms simultaneously**, which is tracked by the property **socket.rooms**.

In the iteration `for (const roomName of allRooms)`, we attempt to identify at least one other room that the socket is in other than its socket ID. If no such room exists (indicating that the socket is not part of any room), we send a notification warning back to the client. Otherwise, we can broadcast this message to all other clients in the same room:
`(nsp.to(currentRoom).emit('message', ….)`

In `socket.on('disconnect', () => {{` callback handler:
we broadcast a message to all connected clients in a given namespace that a particular client has disconnected from that namespace, and also the remaining sockets (connected clients) left in that namespace.
`nsp.emit('message', { …. }`

5.1.2 client.js

Many parts of the code here is similar to the previous example involving namespaces alone. Additional code relevant to the handling of rooms include:

In the previous implementation, we had a callback implemented for user count in a given name space. Now we also have an additional callback related to providing updates on user count in a specific room inside the handler for connecting to namespace:
`connectBtn.addEventListener('click', () => {`

In this callback
`socket.on('roomCountUpdate', (data) => {`
`roomCountUpdate event` - Receives room user count updates from the server.
`UI update` - Displays the number of users currently in that specific room by checking if `(data.room === currentRoom)`

For the event handler related to user clicking on "Join Room"
`joinRoomBtn.addEventListener` - Waits for user to click "Join Room"
`socket.emit('joinRoom', roomName)` - Sends the room name to the server. Triggers the server's `socket.on('joinRoom', …)` handler.
UI updates `(roomUI.style.display = 'none')` - Hides the room selection menu and shows the chat interface.
`roomTitle.textContent` - Updates page to show which room the user joined.

The rest of the code is similar to previously.

6 Specifying and displaying usernames

We can make a further refinement to our existing codebase for namespaces with rooms to make it more user-friendly by incorporating these additional features:

- Clients can explicitly specify the username they wish to use via the webpage UI when they connect to a specific namespace and room.
- The client webpage should show the username of the current connected client as well the namespace and room that they are connected to.

We will continue to work with the existing `server.js` in this project folder `secondapp`

Replace it with the content from `labcode` folder
`server-v6.js`

We will use the existing files in the subfolder `public` in `secondapp`:
`index.html`
`client.js`

Replace it with the content from `labcode / public` folder
`index-v6.html`

`client-v6.js`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

http://localhost:3000/

Open this URL in a couple of separate tabs in the same or different web browser.

All the interactions that we have performed previously can be repeated here, with the addition of additional functionality for selecting a username and having the username and the room being shown in the UI for each interacting user.

Again, notice that if you were to stop the Node server and restart it again after a while, if there are still any remaining active clients running in an open browser tab, the client will automatically reconnect after a while and messages indicating this connection will be shown on both the server log output and client UI. This demonstrates the **automatic reconnection feature of Socket.IO which occurs automatically in the background**.

Nonetheless, the reconnected clients will not be able to exchange messages because the **reconnection attempt is to the previously specified namespace of a Socket.IO connection**. You can try this out by attempting to send messages amongst the reconnected clients.

The reason for this is because in the current implementation, messages are broadcast within a specific room in a namespace, and the client needs to also explicitly rejoin a room within a namespace. This is currently not implemented in code yet, and we will see how to do this in an upcoming example.

Press Ctrl+C to terminate the back-end app when you are done.

6.1 Key points about code

6.1.1 server.js

The main changes are:

Inside the `socket.on('joinRoom', (data) => { ... })` event handler.

The server receives a data object (`data.username` and `data.roomName`) from the client, checks whether the username field exists and is not just empty spaces, and if the client provided a valid name, that name is used as the `displayName`. Otherwise, the server falls back to an automatically assigned temporary name like sports_User1

```
socket.data.username = displayName;
socket.data.roomName = roomName;
```

Socket.IO allows attaching arbitrary metadata to each socket connection via `socket.data`. The server saves the user's chosen username and room name here so it can reuse them later (for chat messages, disconnect messages, etc.).

Inside the `socket.on('chatMessage', (msg) => { })` event handler.

```
const displayName = socket.data.username || tempName;
```

The server now looks up the stored username (`socket.data.username`) when broadcasting messages or handling disconnects, and only uses the auto-generated identifiers (`tempName`) if none is existing (i.e. the user did not specify one).

6.1.2 client.js

The main changes are:

In the event listener for connecting to a namespace,
`connectBtn.addEventListener('click', () => {`

Retrieves the user-entered name from the input field (`usernameInput`), checks if it's empty; if so, shows an alert and stops, otherwise stores it in a variable (`currentUsername`) for later use when joining a room.

```
socket.emit('joinRoom', { roomName : roomName, username:
currentUsername });
```

When the client joins a room, it now sends both `roomName` and `username` in a single object. The server reads this data to assign the correct `displayName`.

```
connectionInfo.textContent =
  `You are logged in as "${currentUsername}" in namespace
"${currentNamespace}", room "${currentRoom}".`;
```

Once the user successfully joins a room, this line updates the paragraph element with ID `connectionInfo`. It dynamically shows:

- The current username
- The namespace (e.g., `/sports`)
- The room (e.g., RoomA)

7 Accessing room info after auto reconnection

We saw earlier that if you were to stop the Node server and restart it again after a while, if there are still any remaining active clients running in an open browser tab, the client will automatically reconnect after a while based on the **automatic reconnection feature of Socket.IO which occurs automatically in the background**.

Nonetheless, the reconnected clients will not be able to exchange messages because the **reconnection attempt is to the previously specified namespace of a Socket.IO connection**. However, in the current implementation, messages are broadcast within a specific room in a namespace, and the client needs to also explicitly rejoin a room within a namespace for this to happen.
However, after server restart:

- The server's memory is wiped (it doesn't remember which client was in which room).
- The socket ID changes (a new socket instance is created on the server) when the reconnect attempt succeeds.
- The room membership (socket.join(roomName)) is lost on the client side.

There are two options to restoring the room name info after reconnection:

a) Prompt-based reconnect - When reconnection occurs, prompt the user to choose a room again (and optionally re-enter username). Requires explicit user action
b) Automatic rejoin - Store the user's previous room, namespace, and username in client-side variables, and automatically rejoin the room after reconnection. This is more ideal for production apps where seamless user experience is desired.

We will demonstrate both options here via changes to be made to `client.js` in the subfolder `public` in `secondapp`:

Replace it with the content from `labcode / public` folder
`client-v7.js`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

http://localhost:3000/

Open this URL in a couple of separate tabs in the same or different web browser.

All the interactions that we have performed previously can be repeated here, test this out.

Now make sure there are still a few active clients running in open browser tabs, then stop the Node server. Notice now there are explicit messages coming out to indicate the detection of the disconnection from any open clients, as well as periodical attempts to reconnect. We have now explicitly included event handler code to demonstrate the **automatic reconnection feature of Socket.IO which occurs automatically in the background**.

After a while, restart the Node server. Notice now that any running clients automatically successfully reconnect to the previous namespace and rooms using the variables that we had retained.

Check the code for `client.js`. Currently we are implementing Option B (automatic rejoin). We can also optionally implement Option A (manual rejoin by prompting user to specify room about

reconnect). Comment / Uncomment the relevant pieces of code in `client.js`, close all tabs and restart the Node server to try out Option A.

With this option A, upon reconnect, user is prompted again to manually enter the room before connection is reestablished.

Play around with this for a while and then press Ctrl+C to terminate the back-end app when you are done.

7.1 Key points about code

7.1.1 client.js

We now additionally have an event handler for reconnection attempt events, so we can display associated reconnection attempts messages:

```
socket.io.on('reconnect_attempt', (attempt) => {
```

For the 2 options available for restoring the room name info after reconnection, both these options run inside the event handler for the successful reconnect event:

```
socket.io.on('reconnect', (attemptNumber) => {
```

Option A – manual prompt rejoin

`alert('Reconnected...')` - Displays a browser popup informing the user that they've reconnected but must rejoin a room.
`roomUI.style.display = 'block';` - Makes the room selection panel visible again (where the user can choose a room from the dropdown).
`chatUI.style.display = 'none';` - Hides the chat interface until the user rejoins a room. This prevents users from typing messages before being part of a room

Option B – Automatic rejoin

Code block starting from: `if (currentRoom && currentUsername) {`

This checks if both the last joined room (currentRoom) and the username (currentUsername) are still stored in memory. These variables were originally set earlier in the script.
If both are available, a message is appended to the chat. This informs the user what's happening behind the scenes. Then the client re-emits the joinRoom event:

```
socket.emit('joinRoom', {
```

This triggers the same `socket.on('joinRoom', ...)` handler on the server as if the user had manually rejoined.
If the variables are missing (e.g., user never joined a room yet), the client shows a message, then it restores the UI to let the user manually choose a room again:
`roomUI.style.display = 'block';`

```
chatUI.style.display = 'none';
```

8 Simulation of real time data streaming of machine metrics

We can simulate a visualization dashboard for displaying real time operational parameters for a machine on the production floor. The visualization at the front end here is performed using basic HTML (via the <canvas> element) and basic CSS style rules. A more complex and detailed visualization would require a specialized JavaScript library such as D3.js, which is beyond the scope of this workshop.

We will continue to work with the existing `server.js` in this project folder `secondapp`

Replace it with the content from `labcode` folder
`server-v8.js`

We will use the existing files in the subfolder `public` in `secondapp`:
`index.html`
`client.js`

And also add a new separate stylesheet:
`styles.css`

Replace it with the content from `labcode / public` folder
`index-v8.html`
`client-v8.js`
`styles-v8.css`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

http://localhost:3000/

You should be able to see the log output of simulated machine operational parameters being sent from the server-side and rendered appropriately on the front-end using 4 metric card panels which are updated in real time. There are also 3 live trend charts at the bottom showing the last 10 readings of RPM, Torque, and Pressure

These operational parameters (RPM, Torque, Pressure) are reflective of typical readings from IoT sensors in real life machine environments.

8.1 Key points about code

The Socket.IO connection portion of the code is pretty straightforward and similar to what we have already seen in previous examples. The main difference here is the visualization at the front-end.

8.1.1 server.js

```
function generateMachineData() {
}
```

This generates the random dummy data in a JSON object format that will be emitted to the front-end:

```
setInterval(() => {
…
io.emit('machineData', data);
…

}, EMIT_INTERVAL_MS);
```

This transmits the JSON object to the front-end periodically with the duration determined by `EMIT_INTERVAL_MS`

8.1.2 index.html

```
<div class="dashboard">
<div class="card" id="torque-card">
```

- Each `<div class="card">` represents one metric's visual card panel.
- The `<p>` elements inside each card display numeric values that are dynamically updated by client.js.
- The id attributes (`rpm-value`, `torque-value`, `etc.`) are DOM hooks that JavaScript uses to update the displayed numbers in real-time.
- The fourth card (`timestamp-card`) separates date and time for visual clarity.
- This section forms the numeric "snapshot" visualization layer.

```
<section class="trend-section">
…
…
  <div class="trend-grid">
    <div class="trend-card">
    <div class="trend-card">
```

- Displays mini-charts for live trends of the three numerical metrics.
- Each `<canvas>` is a blank drawing surface used by the Canvas API (in `client.js`) to render trend lines dynamically.

- `id` attributes again serve as DOM references.
- The `<h3>` labels identify which chart corresponds to which metric.
- This entire `<section>` defines the graphical visualization region of the dashboard.

8.1.3 styles.css

```
.dashboard {
 …
}
```

- Arranges the 4 metric cards into a 2×2 grid.
- Provides consistent spacing `(grid-gap: 20px)` and centers the dashboard.
- This creates the visual structure of the numeric display.

```
.card {
}
.card:hover {
}
.card p {
}
```

- Gives each card a panel-like look with shadows and rounded corners.
- Enlarges the metric values `(font-size: 2em)` for readability.
- Adds a small hover animation for visual polish.

```
@media (max-width: 600px) {
}
```

Ensures the visualization remains usable on smaller screens (cards stack vertically).

8.1.4 client.js

```
function drawLineChart(canvas, data, minY, maxY) {
```

- Uses the Canvas 2D API to draw a simple line graph of each metric.
- The chart automatically scales values between given minY/maxY.
- Clears and redraws the chart every time new data arrives.
- Provides a visual trend rather than static numbers.

9 Extending visualization to incorporate namespaces with rooms

We will now extend the previous application to include the use of the rooms and namespaces feature of Socket.IO in the following manner:

The frontend client connects to Socket.IO with a predefined namespace. This namespace is now subdivided into two rooms that conceptually correspond to live streams from two different Machines (Machine A and Machine B). The user can then select which Machine (room) they wish to receive a live data stream feed from. The backend in turn will now generate two streams of dummy data, each stream meant for a particular Machine (room); with each stream of data in the same format as before (4 parameters: timestamp, RPM, Torque, Pressure encapsulated as properties within a JavaScript object).

We will continue to work with the existing `server.js` in this project folder `secondapp`

Replace it with the content from `labcode` folder
`server-v9.js`

We will use the existing files in the subfolder `public` in `secondapp`:
`index.html`
`client.js`

And also add a new separate stylesheet:
`styles.css`

Replace it with the content from `labcode / public` folder
`index-v9.html`
`client-v9.js`
`styles-v9.css`

Start the back end Express server from the command prompt with:

```
node server.js
```

Load `index.html` by accessing the root route / at:

http://localhost:3000/

The visualization should be identical to the previous example; except now you have an additional choice of selecting which machine (room in a namespace) to obtain your livestream data from.

9.1 Key points about code

The visualization functionality of the code is almost identical to the one explained previously: the main addition is now the functionality related to rooms and namespaces that had seen earlier in the context of a chatroom example.

9.1.1 server.js

```
const machineNamespace = io.of('/machines');
```

This line defines a namespace called `/machines`.

```
socket.on('joinRoom', (roomName) => {
```

```
...
...
});
```

Rooms are logical sub-divisions within a namespace. Clients in different rooms of the same namespace do not receive each other's events. The frontend emits `'joinRoom'` with either `"machineA"` or `"machineB"`.

The server:

- Validates the room name.
- Makes the client leave its previous room (if switching).
- Makes it join the new room.
- This ensures each connected client receives only one stream (A or B).

The variable `currentRoom` tracks which room this socket belongs to.

```
machineNamespace.to('machineA').emit('machineData', dataA);
machineNamespace.to('machineB').emit('machineData', dataB);
```

`.to('roomName')` ensures that:

- Clients in `machineA` receive only dataA.
- Clients in `machineB` receive only dataB.

This demonstrates selective broadcasting inside a namespace.
Because both emissions use the same event name `('machineData')`, the frontend code remains consistent — but receives different values depending on its room.

9.1.2 client.js

```
const socket = io('/machines');
```

Instead of `io()` (which connects to the default `/namespace`), the client now connects to `/machines`.

```
machineNamespace.on('connection', ...)
```

This is the key step that binds this frontend to the new namespace.

```
let currentRoom = 'machineA';
const machineSelect = document.getElementById('machine-select');
const currentMachineLabel = document.getElementById('current-machine-
label');
```

- Tracks which room (machine) the client is currently viewing.
- These DOM elements connect the dropdown selector and the display label to JavaScript logic.

```
function joinRoom(roomName) {
```

- When called, this sends a custom event `'joinRoom'` to the server with the selected room name.
- The server then executes: `socket.join(roomName);`
- The frontend updates its label and resets charts to avoid mixing data from two different machines.
- This function is the core frontend logic for room switching.

```
machineSelect.addEventListener('change', (event) => {
});
```

- Listens for user selection from the `<select>` element.
- When the user switches machines, it triggers `joinRoom()` to send a room join request to the server.
- Without this, all clients would stay in their default room forever.

```
socket.on('connect', () => {
  joinRoom(currentRoom);
});
```

- Ensures that immediately after the connection to the `/machines` namespace is established, the client automatically joins the default room (`machineA`).
- Prevents a scenario where the client connects but receives no data because it hasn't joined any room yet.

10 END