



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

WEB API

Content

1. Overview
2. Storage APIs
3. Real-time APIs
4. Worker APIs

Content

1. Overview
2. Storage APIs
3. Real-time APIs
4. Worker APIs

1. Overview


- A Web API is an application programming interface for the Web.
- All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.
- Web APIs are typically used with JavaScript on client-side (browsers), not on server-side, so they're also called Browser APIs
- Some APIs are not supported by all browsers. Check at <https://caniuse.com>
- This lesson will introduce some popular Web APIs:
 - Storage APIs: IndexedDB, Cache API, Local Storage, Session Storage
 - Real-time APIs: Websockets, Server Sent Events, WebRTC
 - Worker APIs: Web Workers, Service Workers

1. Overview

- Eg: Battery Status API only supported on Google Chrome 38-119, Edge 79-107, Firefox 43-51,...

Battery Status API - CR

Method to provide information about the battery status of the hosting device.

Current aligned Usage relative Date relative Filtered All 							
Chrome	Edge [*]	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS [*]
			2-9				
			10-15				
			16-42				
4-36			43-51	10-24			
37	12-18						
38-106	79-106	3.1-16.0	52-106	25-91	6-10		3.2-16.0
107	107	16.1	107	92	11	107	16.1
108-110		16.2-TP	108-109				

```
> navigator.getBattery().then(b => console.log(b))
▼ BatteryManager {charging: true, chargingTime: 0, dischargingTime: Infinity, Level: 1, onchargingchange: null, ...} ⓘ
  charging: true
  chargingTime: 0
  dischargingTime: Infinity
  level: 1
  onchargingchange: null
  onchargingtimechange: null
  ondischargingtimechange: null
  onlevelchange: null
  ▶ [[Prototype]]: BatteryManager
< ▶ Promise {<fulfilled>: undefined}
> navigator.userAgent
< 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0 Safari/537.36'
```

Google Chrome 107

```
>> navigator.getBattery().then(b => console.log(b))
```

```
! ▶ Uncaught TypeError: navigator.getBattery is not a function
  <anonymous> debugger eval code:1
  [Learn More]
```

```
>> navigator.userAgent
```

```
← "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:106.0) Gecko/20100101 Firefox/106.0"
```

```
>>
```

Firefox 106

Content


1. Overview
2. **Storage APIs**
3. Real-time APIs
4. Worker APIs

2. Storage APIs

2.1. Cookie

- A cookie is a small piece of information left on a visitor's computer by a website, via a web browser.
- Cookies are used to personalize a user's web experience with a website. It may contain the user's preferences or inputs when accessing that website. A user can customize their web browser to accept, reject, or delete cookies.
- Cookies can be set and modified at the server level using the Set-Cookie HTTP header, or with JavaScript using **document.cookie**.
- Cookies is a string containing a semicolon-separated list of all cookies (i.e. key=value pairs)

```
> document.cookie  
⏏ ' _ga=GA1.1.366429479.1665889130; c_c=1; _ga_27CQ8LV57D=GS1.1.1669562734.22.0.1669562734.0.0.0'
```



```
1 // read all cookie  
2 allCookies = document.cookie;  
3 // write a new cookie  
4 document.cookie = newCookie;
```

2. Storage APIs

2.2. Local Storage and Session Storage

- **sessionStorage** stores data for only one session, meaning that the data is stored until the browser (or tab) is closed.
- **localStorage** does the same thing, but persists even when the browser is closed and reopened.
 - Stores data with no expiration date, and gets cleared only through JavaScript, or clearing the Browser cache / Locally Stored Data.

2. Storage APIs

- Local (Session) Storage can be access by **localStorage** (**sessionStorage**) object with 3 methods: **getItem**, **setItem**, **removeItem**

```
> localStorage
< Storage {survey.INTEROP_2023: '{"random":0.41242170761190455,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', theme: 'dark', frequently-viewed-documents: '[{"url":"/en-US/docs/Web/API/Fetch_API","title":"F...stamp":1659461744545,"visitCount":3,"serial":16}]', survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', length: 4}
  frequently-viewed-documents: "[{"url":"/en-US/docs/Web/API/Fetch_API","title":"F...stamp":1659461744545,"visitCount":3,"serial":16}]", survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', length: 4
  survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}'
  survey.INTEROP_2023: '{"random":0.41242170761190455,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}'
  theme: "dark"
  length: 4
  ▶ [[Prototype]]: Storage

> localStorage.setItem("description", "Web APIs")
< undefined

> localStorage
< Storage {description: 'Web APIs', survey.INTEROP_2023: '{"random":0.41242170761190455,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', theme: 'dark', frequently-viewed-documents: '[{"url":"/en-US/docs/Web/API/Fetch_API","title":"F...stamp":1659461744545,"visitCount":3,"serial":16}]', survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', ...}
  description: "Web APIs"
  frequently-viewed-documents: "[{"url":"/en-US/docs/Web/API/Fetch_API","title":"F...stamp":1659461744545,"visitCount":3,"serial":16}]", survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', length: 5
  survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}'
  survey.INTEROP_2023: '{"random":0.41242170761190455,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}'
  theme: "dark"
  length: 5
  ▶ [[Prototype]]: Storage

> localStorage.getItem("description")
< 'Web APIs'

> localStorage.removeItem("description")
< undefined

> localStorage
< Storage {survey.INTEROP_2023: '{"random":0.41242170761190455,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', theme: 'dark', frequently-viewed-documents: '[{"url":"/en-US/docs/Web/API/Fetch_API","title":"F...stamp":1659461744545,"visitCount":3,"serial":16}]', survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', length: 4}
  frequently-viewed-documents: "[{"url":"/en-US/docs/Web/API/Fetch_API","title":"F...stamp":1659461744545,"visitCount":3,"serial":16}]", survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}', length: 4
  survey.BROWSER_SURVEY_OCT_2022: '{"random":0.2629911540447156,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}'
  survey.INTEROP_2023: '{"random":0.41242170761190455,"seen_at":null,"dismissed_at":null,"submitted_at":null,"opened_at":null}'
  theme: "dark"
  length: 4
  ▶ [[Prototype]]: Storage
```

2. Storage APIs

2.3. IndexedDB

- IndexedDB is a low-level API for client-side storage of significant amounts of structured data, including files/blobs. This API uses indexes to enable high-performance searches of this data.
- While Web Storage is useful for storing smaller amounts of data, IndexedDB provides a solution for storing larger amounts of structured data.
- IndexedDB is a transactional database system, like an SQL-based RDBMS. However, unlike SQL-based RDBMSes, which use fixed-column tables, IndexedDB is a JavaScript-based object-oriented database. IndexedDB lets you store and retrieve objects that are indexed with a key.

Content

1. Overview
2. Storage APIs
3. Real-time APIs
4. Worker APIs

3. Real-time APIs

3.1. WebSocket API

- The WebSocket API is an advanced technology that makes it possible to open a two-way interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply (such as HTTP request).
- In order to use WebSocket, we need a WebSocket server. It's simply an application listening on any port of a TCP server that follows a specific protocol. A WebSocket server can be written in any server-side language as Java, C++, Golang,...

3. Real-time APIs

3.1. WebSocket API

- First, the server must listen for incoming socket connections using a standard TCP socket.
- A client can start the WebSocket handshake process by contacting the server and requesting a WebSocket connection. It will send a pretty standard HTTP request with headers that looks like this (the HTTP version must be 1.1 or greater, and the method must be GET):

```
GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

3. Real-time APIs

3.1. WebSocket API

- When server receives the handshake request, it should send back a special response that indicates that the protocol will be changing from HTTP to WebSocket. That header looks something like:

```
HTTP/1.1 101 Switching Protocols  
Upgrade: websocket  
Connection: Upgrade  
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+x0o=
```

3. Real-time APIs

3.1. WebSocket API

```
1 // simple web socket server written in NodeJS
2 const WebSocketServer = require("ws").WebSocketServer;
3
4 const server = new WebSocketServer({ port: 3000 });
5
6 server.on("connection", (socket) => {
7   // send a message to the client
8   socket.send(
9     JSON.stringify({
10       type: "hello from server",
11       content: [1, "2"],
12     })
13   );
14
15   // receive a message from the client
16   socket.on("message", (data) => {
17     const packet = JSON.parse(data);
18
19     switch (packet.type) {
20       case "hello from client":
21         // ...
22         break;
23     }
24   });
25 });
26
```

```
1 const client = new WebSocket("wss://www.example.com:3000");
2
3 function sendText() {
4   const msg = {
5     type: "message",
6     text: document.getElementById("text").value,
7     id: clientID,
8     date: Date.now(),
9   };
10
11   // Send the msg object as a JSON-formatted string.
12   client.send(JSON.stringify(msg));
13 }
14
15 function handle(message) {
16   // handle message receive from server
17   // ...
18 }
19
20 // message event is sent to WebSocket object (client)
21 // when receive messages from server
22 client.onmessage((event) => {
23   handle(event.data);
24 });
25
26 // close the connection
27 client.close();
28
```

3. Real-time APIs

3.2. Server-sent Events

- Similar to WebSocket, but SSE is one-way connection, so you can't send event from client to server.
- Server will define a HTTP endpoint for SSE, when a client send a request to that endpoint, server will write to header of HTTP response with **Content-type** field is **text/event-stream** and **Connection** field is **keep-alive**

3. Real-time APIs

3.2. Server-sent Events



```
1 const http = require("http");
2
3 const server = http.createServer((req, res) => {
4   // Server-sent events endpoint
5   if (req.url === "/events") {
6     // header of http response:
7     // content-type must be event-stream
8     // and connection must be keep-alive
9     res.writeHead(200, {
10      "Content-Type": "text/event-stream",
11      "Cache-Control": "no-cache",
12      Connection: "keep-alive",
13    });
14
15    // write to response a message each second
16    return setInterval(() => {
17      const id = Date.now();
18      const data = `Hello World ${id}`;
19      const message = `retry: ${refreshRate}\nid:${id}\ndata: ${data}\n\n`;
20      res.write(message);
21    }, 1000);
22  }
23 });
24 // server listen on port 3000
25 server.listen(3000);
26
```



```
1 // eventSource object on client-side listen each message from server
2 const eventSource = new EventSource("/events");
3 eventSource.onmessage((event) => {
4   handle(event.data);
5 });
6
```

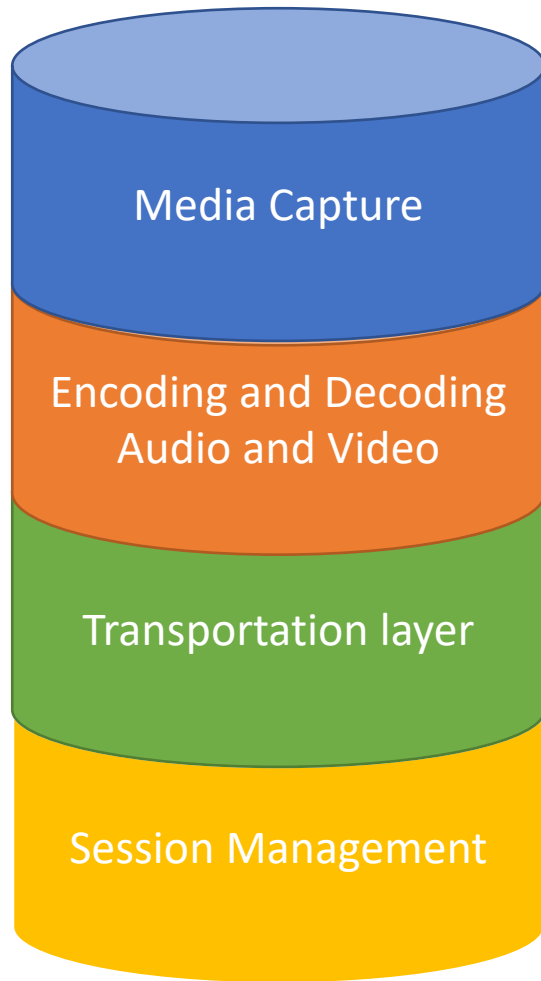
3. Real-time APIs

3.3. WebRTC

- WebRTC (Web Real-Time Communication) is a technology that enables Web applications and sites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary.
- The set of standards that comprise WebRTC makes it possible to share data and perform teleconferencing peer-to-peer, without requiring that the user install plug-ins or any other third-party software.
- Implementations of WebRTC are still evolving, and each browser has different levels of support for codecs and WebRTC features.

3. Real-time APIs

3.3. WebRTC



Media Capture

The first step is to get access to the camera and microphone of the user's device. It detects the type of devices available, get user permission to access these devices and manage the stream.

Encoding and Decoding
Audio and Video

This is the process of splitting up video frames and audio waves into smaller chunks and compressing them in order to send stream of video and audio over internet

Transportation layer

The transportation layer manages the order of packets, deal with packet loss and connecting to other users.

Session Management

The session management deals with managing, opening and organizing connections. This is commonly called **signaling**. If you transfer audio and video streams to the user it also makes sense to transfer collateral data. This is done by the **RTCDDataChannel API**.

Content

1. Overview
2. Storage APIs
3. Real-time APIs
4. **Worker APIs**

4. Worker APIs

- The Javascript code included in HTML document via **<script>** tag runs on main thread. If there is too much activity on the main thread, it can slow down the site, making interactions unresponsive.
- A worker is a script that runs on a thread separate to the browser's main thread.
- There are types of worker: **web workers**, **service workers**, **worklets**. Although they do have some similarities in how they work, they have very little overlap in what they are used for.

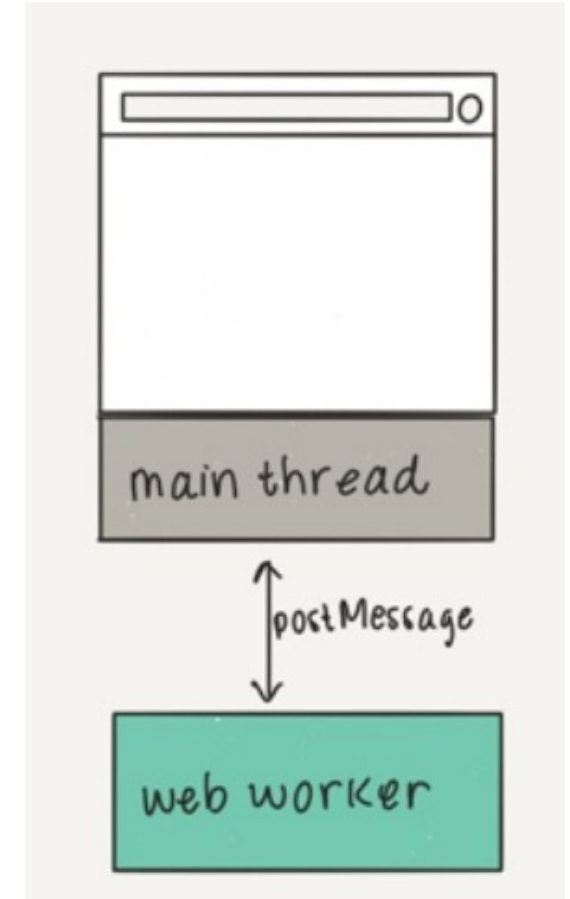
4. Worker APIs

4.1. Web workers

- Web workers are the most general purpose type of worker. They can be used to offload pretty much any heavy processing from the main thread.

eg: image processing

- Main thread and worker can send messages to other with **postMessage** function and listen for messages from other with **onmessage** event.



4. Worker APIs



```
1 /* main.js */
2
3 // Create worker, it will run all code in worker.js
4 const myWorker = new Worker('worker.js');
5
6 // Send message to worker
7 myWorker.postMessage('Hello!');
8
9 // Receive message from worker
10 myWorker.onmessage = function(e) {
11   console.log(e.data);
12 }
```

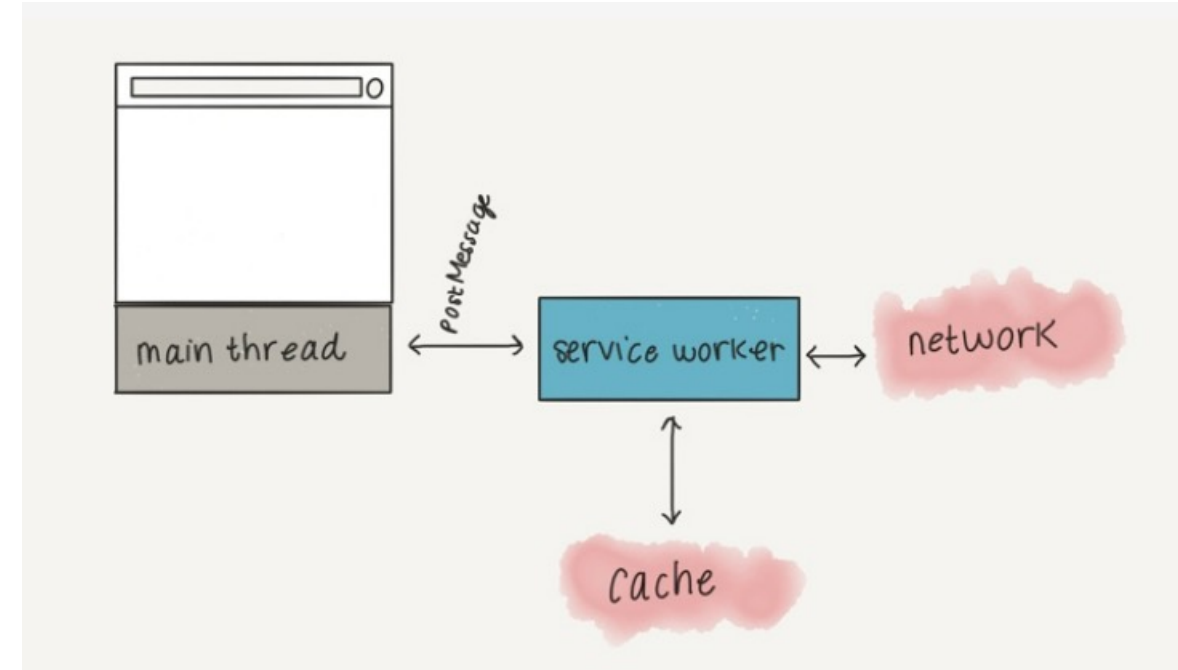


```
1 /* worker.js */
2
3 // Receive message from main file
4 self.onmessage = function(e) {
5   console.log(e.data);
6
7   // Send message to main file
8   self.postMessage(workerResult);
9 }
```

4. Worker APIs

4.2. Service Workers

- Service workers are a type of worker that serve the explicit purpose of being a proxy between the browser and the network and/or cache.
- Like web workers, service workers are registered in the main javascript file, referencing a dedicated service worker file.
- Unlike regular web workers, service workers have some extra features that allow them to fulfil their proxy purpose. Once they are installed and activated, service workers are able to intercept any network requests made from the main document.
- Once intercepted, a service worker can, for example, respond by returning a document from the cache instead of going to the network, thereby allowing web applications to function offline!



4. Worker APIs

The service worker will observe the following lifecycle:

1. Downloaded: when a user first accesses a service worker-controlled site/page.
2. Installed: when the downloaded file is found to be new — either different to an existing service worker (byte-wise compared), or the first service worker encountered for this page/site.
3. Activate: after a successful installation

Service worker is updated when:

- A navigation to an in-scope page occurs.
- An event is fired on the service worker and it hasn't been downloaded in the last 24 hours.



```
1 /* main.js */  
2  
3 navigator.serviceWorker.register('/service-worker.js');
```



```
1 /* service-worker.js */  
2  
3 // Install  
4 self.addEventListener('install', function(event) {  
5     // ...  
6 });  
7  
8 // Activate  
9 self.addEventListener('activate', function(event) {  
10    // ...  
11 });  
12  
13 // Listen for network requests from the main document  
14 self.addEventListener('fetch', function(event) {  
15    // Return data from cache  
16    event.respondWith(  
17        caches.match(event.request);  
18    );  
19 });
```

4. Worker APIs

4.3. Worklet

- The Worklet interface is a lightweight version of Web Workers and gives developers access to low-level parts of the rendering pipeline.
- With Worklets, you can run JavaScript and WebAssembly code to do graphics rendering or audio processing where high performance is required.
- Worklets are restricted to specific use cases; they cannot be used for arbitrary computations like Web Workers. The Worklet interface abstracts properties and methods common to all kind of worklets, and cannot be created directly. We can use:
 - ❑ **PaintWorklet**: For programmatically generating an image where a CSS property expects a file.
Access this interface through **CSS.paintWorklet**
 - ❑ **AudioWorklet**: For audio processing with custom AudioNodes.
 - ❑ **AnimationWorklet**: For creating scroll-linked and other high performance procedural animations.
 - ❑ **LayoutWorklet**: For defining the positioning and dimensions of custom elements.



Exercise 1: Chat app example

- Step 1: Create Project

💡 Command `nodemon index.js` only needs to be typed once and is used to automatically restart the server each time there is a change in the code instead of having to type the command `node index.js` manually multiple times.



```
1 # terminal
2 mkdir chat-example
3 npm init -y
4 npm install express socket.io
5 npm install -D nodemon @faker-js/faker
```

Exercise 1: Chat app example

- Step 2: Init Server in `index.js`

```
1 // index.js
2 const app = require("express")();
3 const http = require("http").Server(app);
4 const io = require("socket.io")(http);
5 // random string generator
6 const { faker } = require("@faker-js/faker");
7
8 const port = 3000;
9
10 app.get("/", (_req, res) => {
11   res.sendFile(__dirname + "/index.html");
12 });
13
14 io.on("connection", (socket) => {
15   // ...
16 });
17
18 http.listen(port, () => {
19   console.log(`Server running at http://localhost:${port}/`);
20 });
21
```

Exercise 1: Chat app example

- Step 3: Init Client in `index.html`

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Socket.IO chat</title>
5     <style>
6       ...
7     </style>
8   </head>
9
10  <body>
11    <h1 id="yourname"></h1>  <!-- display name -->
12    <ul id="messages"></ul>  <!-- display list messages -->
13    <form id="form" action="">
14      <input id="input" autocomplete="off" /><button>Send</button>
15    </form>
16    <!-- load socket.io script from node_modules -->
17    <script src="/socket.io/socket.io.js"></script>
18  </body>
19 </html>
```

Exercise 1: Chat app example

- Step 4: Add socket handler in Client and Server

```
1 // client
2 <script>
3   const socket = io();
4   const messages = document.getElementById("messages");
5   const form = document.getElementById("form");
6   const input = document.getElementById("input");
7   const name = document.getElementById("yourname");
8
9   // send msg to server
10  form.addEventListener("submit", (e) => {
11    e.preventDefault();
12    if (input.value) {
13      socket.emit("chat", input.value);
14      input.value = "";
15    }
16  });
17  // receive msg from server
18  socket.on("sendMsg", (from, msg) => {
19    const item = document.createElement("li");
20    item.textContent = `${from}: ${msg}`;
21    messages.appendChild(item);
22    window.scrollTo(0, document.body.scrollHeight);
23  });
24 </script>
```

```
1 // server
2 io.on("connection", (socket) => {
3   // generate a fake name
4   const username = faker.person.firstName();
5
6   // server receive msg
7   socket.on("chat", (msg) => {
8     // server broadcasts (forward) message to all other clients
9     io.emit("sendMsg", username, msg);
10  });
11 });
```

Exercise 1: Chat app example

- Explain:

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

// This is the server

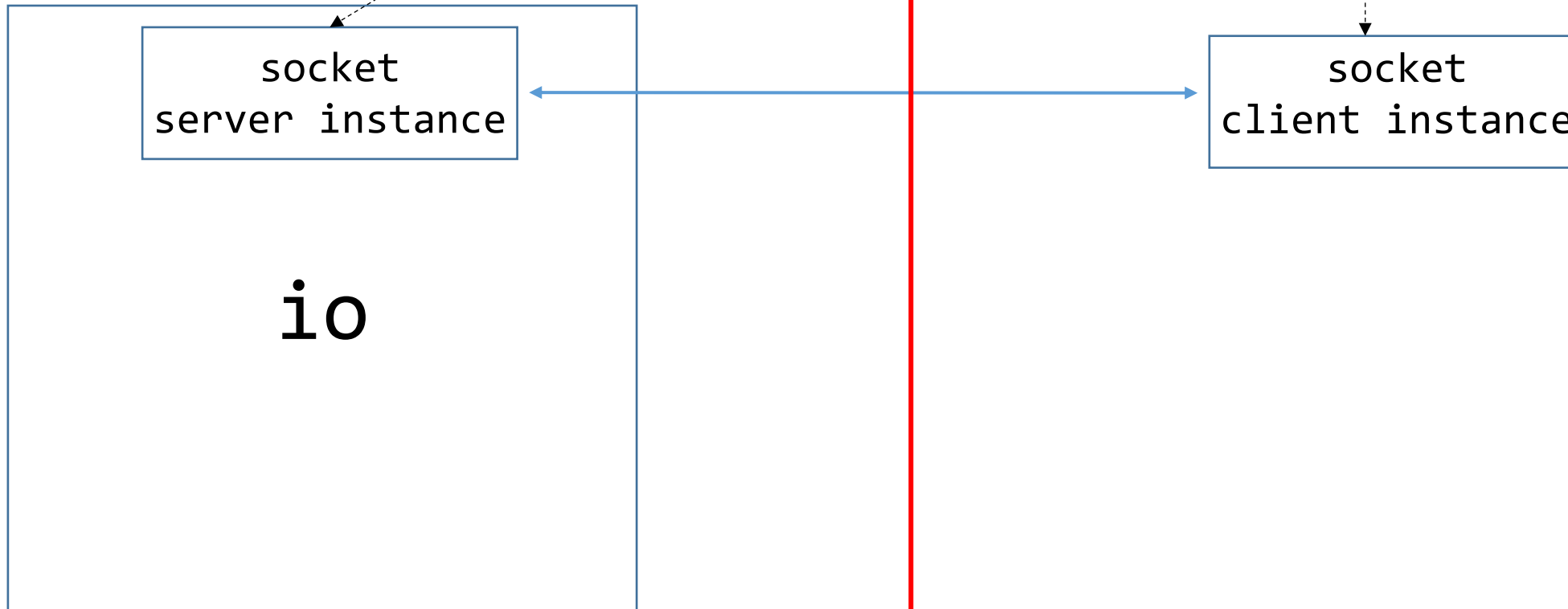
io

Exercise 1: Chat app example

- Explain:

in server: `io.on("connection", (socket) => {...})`

in client: `const socket = io(url);`

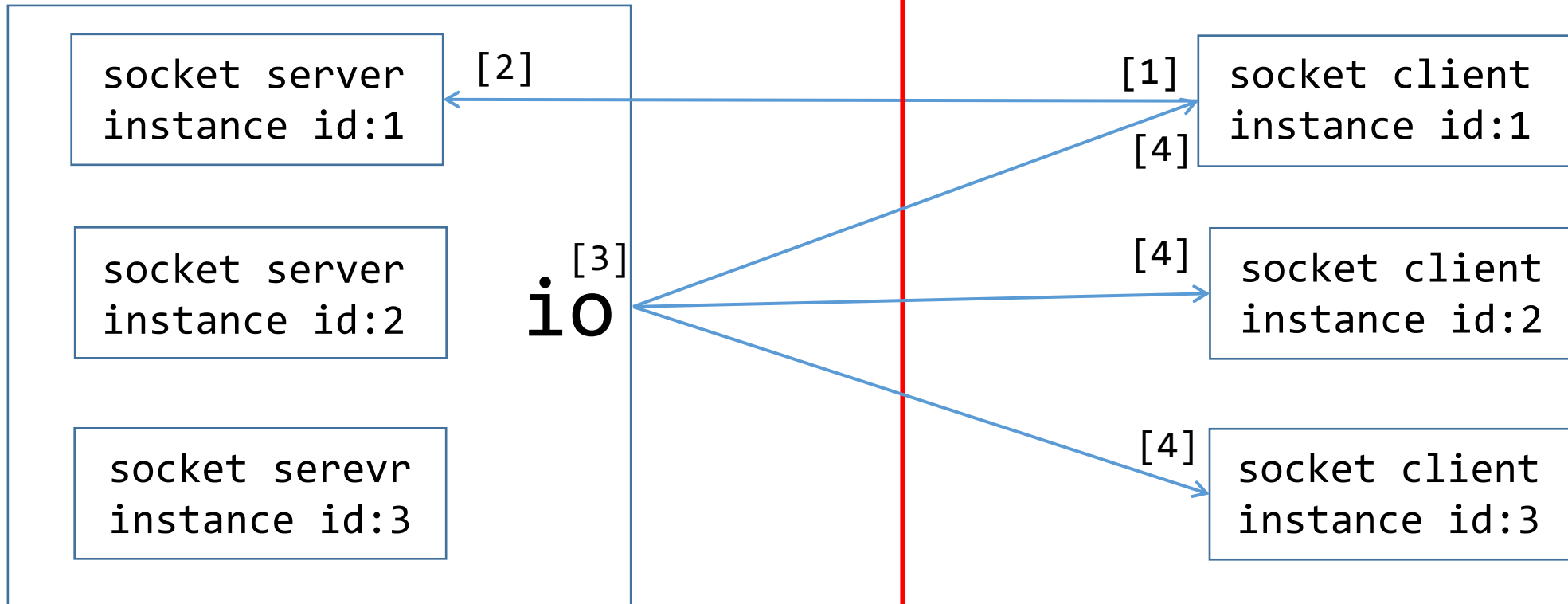


Exercise 1: Chat app example

- Explain:

```
In server: socket.on("chat", msg => { // [2]
  io.emit("sendMsg", username, msg) // [3]
})
```

```
In client: socket.emit("chat", input.value) // [1]
socket.on("sendMsg", (username, msg) => {}) // [4]
```



Exercise 1: Chat app example

- Explain:

In server: `socket.emit("welcome", username) //[1]`

In client: `socket.on("welcome", username => {}) //[2]`

