Version: 4.x

Client API

IO

The `io` method is bound to the global scope in the standalone build:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  const socket = io();
</script>
```

An ESM bundle is also available since version 4.3.0:

```
<script type="module">
  import { io } from "https://cdn.socket.io/4.8.1/socket.io.esm.min.js";

  const socket = io();
</script>
```

With an import map:

```
<script type="importmap">
  {
    "imports": {
      "socket.io-client": "https://cdn.socket.io/4.8.1/socket.io.esm.min.js"
    }
  }
</script>

<script type="module">
  import { io } from "socket.io-client";

  const socket = io();
</script>
```

Else, in all other cases (with some build tools, in Node.js or React Native), it can be imported from the `socket.io-client` package:

```
// ES modules
import { io } from "socket.io-client";

// CommonJS
const { io } = require("socket.io-client");
```

io.protocol

- `<number>`

The protocol revision number (currently: 5).

The protocol defines the format of the packets exchanged between the client and the server. Both the client and the server must use the same revision in order to understand each other.

You can find more information here.

io([url][, options])

- `url` `<string>` (defaults to `window.location.host`)
- `options` `<Object>`
  - `forceNew` `<boolean>` whether to create a new connection
- **Returns** `<Socket>`

Creates a new `Manager` for the given URL, and attempts to reuse an existing `Manager` for subsequent calls, unless the `multiplex` option is passed with `false`. Passing this option is the equivalent of passing `"force new connection": true` or `forceNew: true`.

A new `Socket` instance is returned for the namespace specified by the pathname in the URL, defaulting to `/`. For example, if the `url` is `http://localhost/users`, a transport connection will be established to `http://localhost` and a Socket.IO connection will be established to `/users`.

Query parameters can also be provided, either with the `query` option or directly in the url (example: `http://localhost/users?token=abc`).

To understand what happens under the hood, the following example:

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

const socket = io("ws://example.com/my-namespace", {
  reconnectionDelayMax: 10000,
  auth: {
    token: "123"
  },
  query: {
    "my-key": "my-value"
  }
});
```

is the short version of:

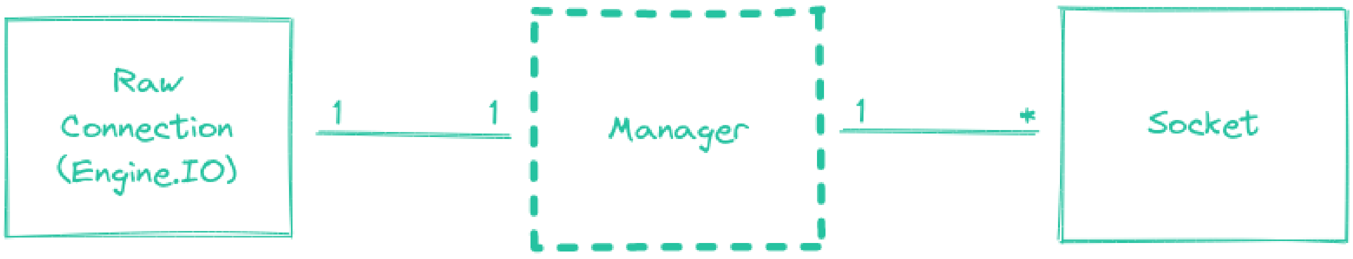
```
import { Manager } from "socket.io-client";

const manager = new Manager("ws://example.com", {
  reconnectionDelayMax: 10000,
  query: {
    "my-key": "my-value"
  }
});

const socket = manager.socket("/my-namespace", {
  auth: {
    token: "123"
  }
});
```

The complete list of available options can be found here.

Manager



The `Manager` *manages* the Engine.IO client instance, which is the low-level engine that establishes the connection to the server (by using transports like WebSocket or HTTP long-polling).

The `Manager` handles the reconnection logic.

A single `Manager` can be used by several Sockets. You can find more information about this multiplexing feature here.

Please note that, in most cases, you won't use the Manager directly but use the Socket instance instead.

Constructor

new Manager(url[, options])

- `url` `<string>`
- `options` `<Object>`
- **Returns** `<Manager>`

The complete list of available options can be found here.

```
import { Manager } from "socket.io-client";

const manager = new Manager("https://example.com");

const socket = manager.socket("/"); // main namespace
const adminSocket = manager.socket("/admin"); // admin namespace
```

Events

Event: 'error'

- `error` `<Error>` error object

Fired upon a connection error.

```
socket.io.on("error", (error) => {
  // ...
});
```

Event: 'ping'

Fired when a ping packet is received from the server.

```
socket.io.on("ping", () => {
  // ...
});
```

Event: 'reconnect'

- `attempt` `<number>` reconnection attempt number

Fired upon a successful reconnection.

```
socket.io.on("reconnect", (attempt) => {
  // ...
});
```

Event: 'reconnect_attempt'

- `attempt` `<number>` reconnection attempt number

Fired upon an attempt to reconnect.

```
socket.io.on("reconnect_attempt", (attempt) => {
  // ...
});
```

Event: 'reconnect_error'

- `error` `<Error>` error object

Fired upon a reconnection attempt error.

```
socket.io.on("reconnect_error", (error) => {
  // ...
});
```

Event: 'reconnect_failed'

Fired when couldn't reconnect within `reconnectionAttempts`.

```
socket.io.on("reconnect_failed", () => {
  // ...
});
```

Methods

manager.connect([callback])

Synonym of manager.open([callback]).

manager.open([callback])

- `callback` `<Function>`
- **Returns** `<Manager>`

If the manager was initiated with `autoConnect` to `false`, launch a new connection attempt.

The `callback` argument is optional and will be called once the attempt fails/succeeds.

```
import { Manager } from "socket.io-client";

const manager = new Manager("https://example.com", {
  autoConnect: false
});

const socket = manager.socket("/");

manager.open((err) => {
  if (err) {
    // an error has occurred
  } else {
    // the connection was successfully established
  }
});
```

manager.reconnection([value])

- `value` `<boolean>`
- **Returns** `<Manager>` | `<boolean>`

Sets the `reconnection` option, or returns it if no parameters are passed.

manager.reconnectionAttempts([value])

- `value` `<number>`
- **Returns** `<Manager>` | `<number>`

Sets the `reconnectionAttempts` option, or returns it if no parameters are passed.

manager.reconnectionDelay([value])

- `value` `<number>`
- **Returns** `<Manager>` | `<number>`

Sets the `reconnectionDelay` option, or returns it if no parameters are passed.

manager.reconnectionDelayMax([value])

- `value` `<number>`
- **Returns** `<Manager>` | `<number>`

Sets the `reconnectionDelayMax` option, or returns it if no parameters are passed.

manager.socket(nsp, options)

- `nsp` `<string>`
- `options` `<Object>`
- **Returns** `<Socket>`

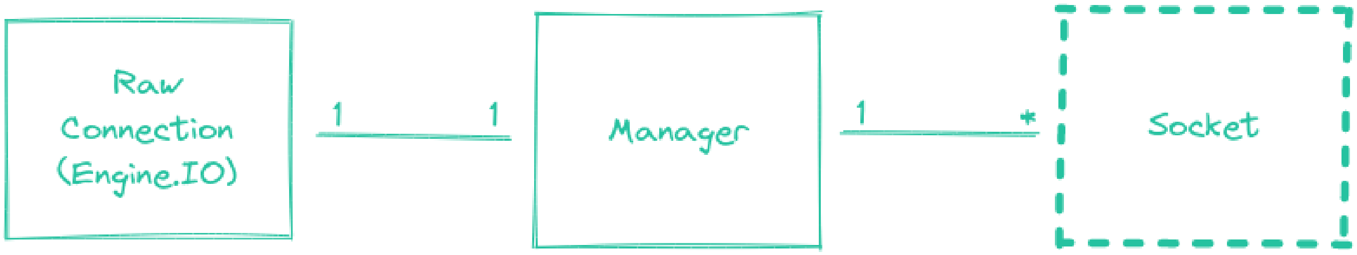
Creates a new `Socket` for the given namespace. Only `auth` (`{ auth: {key: "value"} }`) is read from the `options` object. Other keys will be ignored and should be passed when instancing a `new Manager(nsp, options)`.

manager.timeout([value])

- `value` `<number>`
- **Returns** `<Manager>` | `<number>`

Sets the `timeout` option, or returns it if no parameters are passed.

Socket



A `Socket` is the fundamental class for interacting with the server. A `Socket` belongs to a certain Namespace (by default `/`) and uses an underlying Manager to communicate.

A `Socket` is basically an EventEmitter which sends events to — and receive events from — the server over the network.

```
socket.emit("hello", { a: "b", c: [] });

socket.on("hey", (...args) => {
  // ...
});
```

More information can be found here.

Events

Event: 'connect'

This event is fired by the Socket instance upon connection **and** reconnection.

```
socket.on("connect", () => {
  // ...
});
```

⚠ CAUTION

Event handlers shouldn't be registered in the `connect` handler itself, as a new handler will be registered every time the socket instance reconnects:

BAD ⚠

```
socket.on("connect", () => {
  socket.on("data", () => { /* ... */ });
});
```

GOOD 👍

```
socket.on("connect", () => {
  // ...
});

socket.on("data", () => { /* ... */ });
```

Event: 'connect_error'

- `error` `<Error>`

This event is fired upon connection failure.

Reason	Automatic reconnection?
The low-level connection cannot be established (temporary failure)	✅ YES
The connection was denied by the server in a middleware function	❌ NO

The `socket.active` attribute indicates whether the socket will automatically try to reconnect after a small randomized delay:

```
socket.on("connect_error", (error) => {
  if (socket.active) {
    // temporary failure, the socket will automatically try to reconnect
  } else {
    // the connection was denied by the server
    // in that case, `socket.connect()` must be manually called in order to
reconnect
    console.log(error.message);
  }
});
```

Event: 'disconnect'

- `reason` `<string>`

- `details` `<DisconnectDetails>`

This event is fired upon disconnection.

```
socket.on("disconnect", (reason, details) => {
  // ...
});
```

Here is the list of possible reasons:

Reason	Description	Automatic reconnection?
`io server disconnect`	The server has forcefully disconnected the socket with socket.disconnect()	❌ NO
`io client disconnect`	The socket was manually disconnected using socket.disconnect()	❌ NO
`ping timeout`	The server did not send a PING within the `pingInterval + pingTimeout` range	✅ YES
`transport close`	The connection was closed (example: the user has lost connection, or the network was changed from WiFi to 4G)	✅ YES
`transport error`	The connection has encountered an error (example: the server was killed during a HTTP long-polling cycle)	✅ YES

The `socket.active` attribute indicates whether the socket will automatically try to reconnect after a small randomized delay:

```
socket.on("disconnect", (reason) => {
  if (socket.active) {
    // temporary disconnection, the socket will automatically try to reconnect
  } else {
    // the connection was forcefully closed by the server or the client itself
    // in that case, `socket.connect()` must be manually called in order to
reconnect
```

```
    console.log(reason);
  }
});
```

Attributes

socket.active

- `<boolean>`

Whether the socket will automatically try to reconnect.

This attribute can be used after a connection failure:

```
socket.on("connect_error", (error) => {
  if (socket.active) {
    // temporary failure, the socket will automatically try to reconnect
  } else {
    // the connection was denied by the server
    // in that case, `socket.connect()` must be manually called in order to reconnect
    console.log(error.message);
  }
});
```

Or after a disconnection:

```
socket.on("disconnect", (reason) => {
  if (socket.active) {
    // temporary disconnection, the socket will automatically try to reconnect
  } else {
    // the connection was forcefully closed by the server or the client itself
    // in that case, `socket.connect()` must be manually called in order to reconnect
    console.log(reason);
  }
});
```

socket.connected

- `<boolean>`

Whether the socket is currently connected to the server.

```
const socket = io();

console.log(socket.connected); // false

socket.on("connect", () => {
  console.log(socket.connected); // true
});
```

socket.disconnected

- `<boolean>`

Whether the socket is currently disconnected from the server.

```
const socket = io();

console.log(socket.disconnected); // true

socket.on("connect", () => {
  console.log(socket.disconnected); // false
});
```

socket.id

- `<string>`

A unique identifier for the socket session. Set after the `connect` event is triggered, and updated after the `reconnect` event.

```
const socket = io();

console.log(socket.id); // undefined

socket.on("connect", () => {
  console.log(socket.id); // "G5p5..."
});
```

CAUTION

The `id` attribute is an **ephemeral** ID that is not meant to be used in your application (or only for debugging purposes) because:

- this ID is regenerated after each reconnection (for example when the WebSocket connection is severed, or when the user refreshes the page)
- two different browser tabs will have two different IDs
- there is no message queue stored for a given ID on the server (i.e. if the client is disconnected, the messages sent from the server to this ID are lost)

Please use a regular session ID instead (either sent in a cookie, or stored in the localStorage and sent in the `auth` payload).

See also:

- Part II of our private message guide
- How to deal with cookies

socket.io

- `<Manager>`

A reference to the underlying Manager.

```
socket.on("connect", () => {
  const engine = socket.io.engine;
  console.log(engine.transport.name); // in most cases, prints "polling"

  engine.once("upgrade", () => {
    // called when the transport is upgraded (i.e. from HTTP long-polling to
WebSocket)
    console.log(engine.transport.name); // in most cases, prints "websocket"
  });

  engine.on("packet", ({ type, data }) => {
    // called for each packet received
  });

  engine.on("packetCreate", ({ type, data }) => {
    // called for each packet sent
  });

  engine.on("drain", () => {
    // called when the write buffer is drained
  });

  engine.on("close", (reason) => {
    // called when the underlying connection is closed
  });
});
```

socket.recovered

Added in v4.6.0

- `<boolean>`

Whether the connection state was successfully recovered during the last reconnection.

```
socket.on("connect", () => {
  if (socket.recovered) {
    // any event missed during the disconnection period will be received now
  } else {
    // new or unrecoverable session
  }
});
```

More information about this feature here.

Methods

socket.close()

Added in v1.0.0

Synonym of socket.disconnect().

socket.compress(value)

- `value` `<boolean>`
- **Returns** `<Socket>`

Sets a modifier for a subsequent event emission that the event data will only be *compressed* if the value is `true`. Defaults to `true` when you don't call the method.

```
socket.compress(false).emit("an event", { some: "data" });
```

socket.connect()

Added in v1.0.0

- **Returns** `Socket`

Manually connects the socket.

```
const socket = io({
  autoConnect: false
});

// ...
socket.connect();
```

It can also be used to manually reconnect:

```
socket.on("disconnect", () => {
  socket.connect();
});
```

socket.disconnect()

Added in v1.0.0

- **Returns** `<Socket>`

Manually disconnects the socket. In that case, the socket will not try to reconnect.

Associated disconnection reason:

- client-side: `"io client disconnect"`
- server-side: `"client namespace disconnect"`

If this is the last active Socket instance of the Manager, the low-level connection will be closed.

socket.emit(eventName[, ...args][, ack])

- `eventName` `<string>` | `<symbol>`

- `args` `<any[]>`
- `ack` `<Function>`
- **Returns** `true`

Emits an event to the socket identified by the string name. Any other parameters can be included. All serializable data structures are supported, including `Buffer`.

```
socket.emit("hello", "world");
socket.emit("with-binary", 1, "2", { 3: "4", 5: Buffer.from([6, 7, 8]) });
```

The `ack` argument is optional and will be called with the server answer.

Client

```
socket.emit("hello", "world", (response) => {
  console.log(response); // "got it"
});
```

Server

```
io.on("connection", (socket) => {
  socket.on("hello", (arg, callback) => {
    console.log(arg); // "world"
    callback("got it");
  });
});
```

socket.emitWithAck(eventName[, ...args])

Added in v4.6.0

- `eventName` `<string>` | `<symbol>`
- `args` `any[]`
- **Returns** `Promise<any>`

Promised-based version of emitting and expecting an acknowledgement from the server:

```
// without timeout
const response = await socket.emitWithAck("hello", "world");
```

```
// with a specific timeout
try {
  const response = await socket.timeout(10000).emitWithAck("hello", "world");
} catch (err) {
  // the server did not acknowledge the event in the given delay
}
```

The example above is equivalent to:

```
// without timeout
socket.emit("hello", "world", (val) => {
  // ...
});

// with a specific timeout
socket.timeout(10000).emit("hello", "world", (err, val) => {
  // ...
});
```

And on the receiving side:

```
io.on("connection", (socket) => {
  socket.on("hello", (arg1, callback) => {
    callback("got it"); // only one argument is expected
  });
});
```

CAUTION

Environments that do not support Promises will need to add a polyfill in order to use this feature.

socket.listeners(eventName)

Inherited from the EventEmitter class.

- `eventName` `<string>` | `<symbol>`
- **Returns** `<Function[]>`

Returns the array of listeners for the event named `eventName`.

```
socket.on("my-event", () => {
  // ...
});

console.log(socket.listeners("my-event")); // prints [ [Function] ]
```

socket.listenersAny()

Added in v3.0.0

- **Returns** `<Function[]>`

Returns the list of registered catch-all listeners.

```
const listeners = socket.listenersAny();
```

socket.listenersAnyOutgoing()

Added in v4.5.0

- **Returns** `<Function[]>`

Returns the list of registered catch-all listeners for outgoing packets.

```
const listeners = socket.listenersAnyOutgoing();
```

socket.off([eventName][, listener])

Inherited from the EventEmitter class.

- `eventName` `<string>` | `<symbol>`
- `listener` `<Function>`
- **Returns** `<Socket>`

Removes the specified `listener` from the listener array for the event named `eventName`.

```
const myListener = () => {
  // ...
}
```

```
socket.on("my-event", myListener);

// then later
socket.off("my-event", myListener);
```

The `listener` argument can also be omitted:

```
// remove all listeners for that event
socket.off("my-event");

// remove all listeners for all events
socket.off();
```

socket.offAny([listener])

Added in v3.0.0

- `listener` `<Function>`

Removes the previously registered listener. If no listener is provided, all catch-all listeners are removed.

```
const myListener = () => { /* ... */ };

socket.onAny(myListener);

// then, later
socket.offAny(myListener);

socket.offAny();
```

socket.offAnyOutgoing([listener])

Added in v4.5.0

- `listener` `<Function>`

Removes the previously registered listener. If no listener is provided, all catch-all listeners are removed.

```
const myListener = () => { /* ... */ };
```

```
socket.onAnyOutgoing(myListener);

// remove a single listener
socket.offAnyOutgoing(myListener);

// remove all listeners
socket.offAnyOutgoing();
```

socket.on(eventName, callback)

Inherited from the EventEmitter class.

- `eventName` `<string>` | `<symbol>`
- `listener` `<Function>`
- **Returns** `<Socket>`

Register a new handler for the given event.

```
socket.on("news", (data) => {
  console.log(data);
});

// with multiple arguments
socket.on("news", (arg1, arg2, arg3, arg4) => {
  // ...
});
// with callback
socket.on("news", (cb) => {
  cb(0);
});
```

socket.onAny(callback)

Added in v3.0.0

- `callback` `<Function>`

Register a new catch-all listener.

```
socket.onAny((event, ...args) => {
  console.log(`got ${event}`);
});
```

CAUTION

Acknowledgements are not caught in the catch-all listener.

```
socket.emit("foo", (value) => {
  // ...
});

socket.onAnyOutgoing(() => {
  // triggered when the event is sent
});

socket.onAny(() => {
  // not triggered when the acknowledgement is received
});
```

socket.onAnyOutgoing(callback)

Added in v4.5.0

- `callback` `<Function>`

Register a new catch-all listener for outgoing packets.

```
socket.onAnyOutgoing((event, ...args) => {
  console.log(`got ${event}`);
});
```

CAUTION

Acknowledgements are not caught in the catch-all listener.

```
socket.on("foo", (value, callback) => {
  callback("OK");
});

socket.onAny(() => {
  // triggered when the event is received
});

socket.onAnyOutgoing(() => {
  // not triggered when the acknowledgement is sent
});
```

socket.once(eventName, callback)

Inherited from the EventEmitter class.

- `eventName` `<string>` | `<symbol>`
- `listener` `<Function>`
- **Returns** `<Socket>`

Adds a one-time `listener` function for the event named `eventName`. The next time `eventName` is triggered, this listener is removed and then invoked.

```
socket.once("my-event", () => {
  // ...
});
```

socket.open()

Added in v1.0.0

Synonym of socket.connect().

socket.prependAny(callback)

Added in v3.0.0

- `callback` `<Function>`

Register a new catch-all listener. The listener is added to the beginning of the listeners array.

```
socket.prependAny((event, ...args) => {
  console.log(`got ${event}`);
});
```

socket.prependAnyOutgoing(callback)

Added in v4.5.0

- `callback` `<Function>`

Register a new catch-all listener for outgoing packets. The listener is added to the beginning of the listeners array.

```
socket.prependAnyOutgoing((event, ...args) => {
  console.log(`got ${event}`);
});
```

socket.send([...args][, ack])

- `args` `<any[]>`
- `ack` `<Function>`
- **Returns** `<Socket>`

Sends a `message` event. See socket.emit(eventName[, ...args][, ack]).

socket.timeout(value)

Added in v4.4.0

- `value` `<number>`
- **Returns** `<Socket>`

Sets a modifier for a subsequent event emission that the callback will be called with an error when the given number of milliseconds have elapsed without an acknowledgement from the server:

```
socket.timeout(5000).emit("my-event", (err) => {
  if (err) {
    // the server did not acknowledge the event in the given delay
  }
});
```

Flags

Flag: 'volatile'

Added in v3.0.0

Sets a modifier for the subsequent event emission indicating that the packet may be dropped if:

- the socket is not connected
- the low-level transport is not writable (for example, when a `POST` request is already running in HTTP long-polling mode)

```
socket.volatile.emit(/* ... */); // the server may or may not receive it
```

Edit this page

*Last updated on **Aug 22, 2025***