

[Home](#) [Events](#) [Emitting events](#)

Version: 4.x

# Emitting events

There are several ways to send events between the server and the client.

**TIP**

For TypeScript users, it is possible to provide type hints for the events. Please check [this](#).

## Basic emit

The Socket.IO API is inspired from the Node.js [EventEmitter](#), which means you can emit events on one side and register listeners on the other:

### Server

```
io.on("connection", (socket) => {  
  socket.emit("hello", "world");  
});
```

### Client

```
socket.on("hello", (arg) => {  
  console.log(arg); // world  
});
```

This also works in the other direction:

### Server

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

### Client

```
socket.emit("hello", "world");
```

You can send any number of arguments, and all serializable data structures are supported, including binary objects like [Buffer](#) or [TypedArray](#).

### Server

```
io.on("connection", (socket) => {  
  socket.emit("hello", 1, "2", { 3: '4', 5: Buffer.from([6]) });  
});
```

### Client

```
// client-side  
socket.on("hello", (arg1, arg2, arg3) => {  
  console.log(arg1); // 1  
  console.log(arg2); // "2"  
  console.log(arg3); // { 3: '4', 5: ArrayBuffer (1) [ 6 ] }  
});
```

There is no need to run `JSON.stringify()` on objects as it will be done for you.

```
// BAD  
socket.emit("hello", JSON.stringify({ name: "John" }));  
  
// GOOD  
socket.emit("hello", { name: "John" });
```



### Notes:

- [Date](#) objects will be converted to (and received as) their string representation, e.g. `1970-01-01T00:00:00.000Z`
- [Map](#) and [Set](#) must be manually serialized:

```
const serializedMap = [...myMap.entries()];  
const serializedSet = [...mySet.keys()];
```

- you can use the `toJSON()` method to customize the serialization of an object

Example with a class:

```
class Hero {
  #hp;

  constructor() {
    this.#hp = 42;
  }

  toJSON() {
    return { hp: this.#hp };
  }
}

socket.emit("here's a hero", new Hero());
```

## Acknowledgements

Events are great, but in some cases you may want a more classic request-response API. In Socket.IO, this feature is named acknowledgements.

You can add a callback as the last argument of the `emit()`, and this callback will be called once the other side acknowledges the event:

*Server*

```
io.on("connection", (socket) => {
  socket.on("update item", (arg1, arg2, callback) => {
    console.log(arg1); // 1
    console.log(arg2); // { name: "updated" }
    callback({
      status: "ok"
    });
  });
});
```

*Client*

```
socket.emit("update item", "1", { name: "updated" }, (response) => {  
  console.log(response.status); // ok  
});
```

## With timeout

Starting with Socket.IO v4.4.0, you can now assign a timeout to each emit:

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

You can also use both a timeout and an [acknowledgement](#):

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

## Volatile events

Volatile events are events that will not be sent if the underlying connection is not ready (a bit like [UDP](#), in terms of reliability).

This can be interesting for example if you need to send the position of the characters in an online game (as only the latest values are useful).

```
socket.volatile.emit("hello", "might or might not be received");
```

Another use case is to discard events when the client is not connected (by default, the events are buffered until reconnection).

Example:

### Server

```
io.on("connection", (socket) => {  
  console.log("connect");  
  
  socket.on("ping", (count) => {  
    console.log(count);  
  });  
});
```

### Client

```
let count = 0;  
setInterval(() => {  
  socket.volatile.emit("ping", ++count);  
}, 1000);
```


If you restart the server, you will see in the console:

```
connect  
1  
2  
3  
4  
# the server is restarted, the client automatically reconnects  
connect  
9  
10  
11
```

Without the `volatile` flag, you would see:

```
connect  
1  
2  
3  
4  
# the server is restarted, the client automatically reconnects and sends i  
buffered events  
connect
```

5  
6  
7  
8  
9  
10  
11

 [Edit this page](#)

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