Fork me on GitHub
(https://github.com/socketio/socket.io-client-java)

# Initialization

**Table of content**

# Creation of a Socket instance

```
URI uri = URI.create("https://example.com");
IO.Options options = IO.Options.builder()
        // ...
        .build();

Socket socket = IO.socket(uri, options);
```

Unlike the JS client (which can infer it from the `window.location` object), the URI is mandatory here.

The scheme (https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Syntax) part of the URI is also mandatory. Both `ws://` and `http://` can be used interchangeably.

```
Socket socket = IO.socket("https://example.com"); // OK
Socket socket = IO.socket("wss://example.com"); // OK, similar to the example above
Socket socket = IO.socket("192.168.0.1:1234"); // NOT OK, missing the scheme part
```

The path represents the Namespace (https://socket.io/docs/v4/namespaces/) , and not the actual path (see below) of the HTTP requests:

```
Socket socket = IO.socket(URI.create("https://example.com")); // the main namespace
Socket productSocket = IO.socket(URI.create("https://example.com/product")); // the "product" namespac
e
Socket orderSocket = IO.socket(URI.create("https://example.com/order")); // the "order" namespace
```

# Default values

```
IO.Options options = IO.Options.builder()
    // IO factory options
    .setForceNew(false)
    .setMultiplex(true)

    // low-level engine options
    .setTransports(new String[] { Polling.NAME, WebSocket.NAME })
    .setUpgrade(true)
    .setRememberUpgrade(false)
    .setPath("/socket.io/")
    .setQuery(null)
    .setExtraHeaders(null)

    // Manager options
    .setReconnection(true)
    .setReconnectionAttempts(Integer.MAX_VALUE)
    .setReconnectionDelay(1_000)
    .setReconnectionDelayMax(5_000)
    .setRandomizationFactor(0.5)
    .setTimeout(20_000)

    // Socket options
    .setAuth(null)
    .build();
```

# Description

## IO factory options

These settings will be shared by all Socket instances attached to the same Manager.

`forceNew`

Default value: `false`

Whether to create a new Manager instance.

A Manager instance is in charge of the low-level connection to the server (established with HTTP long-polling or WebSocket). It handles the reconnection logic.

A Socket instance is the interface which is used to sends events to — and receive events from — the server. It belongs to a given namespace (https://socket.io/docs/v4/namespaces) .

A single Manager can be attached to several Socket instances.

The following example will reuse the same Manager instance for the 3 Socket instances (one single WebSocket connection):

```
IO.Options options = IO.Options.builder()
        .setForceNew(false)
        .build();

Socket socket = IO.socket(URI.create("https://example.com"), options); // the main namespace
Socket productSocket = IO.socket(URI.create("https://example.com/product"), options); // the "product"
namespace
Socket orderSocket = IO.socket(URI.create("https://example.com/order"), options); // the "order" names
pace
```

The following example will create 3 different Manager instances (and thus 3 distinct WebSocket connections):

```
IO.Options options = IO.Options.builder()
        .setForceNew(true)
        .build();

Socket socket = IO.socket(URI.create("https://example.com"), options); // the main namespace
Socket productSocket = IO.socket(URI.create("https://example.com/product"), options); // the "product"
namespace
Socket orderSocket = IO.socket(URI.create("https://example.com/order"), options); // the "order" names
pace
```

### `multiplex`

Default value: `true`

The opposite of `forceNew`: whether to reuse an existing Manager instance.

## Low-level engine options

### `transports`

Default value: `new String[] { Polling.NAME, WebSocket.NAME }`

The low-level connection to the Socket.IO server can either be established with:

- HTTP long-polling: successive HTTP requests (`POST` for writing, `GET` for reading)
- WebSocket (https://en.wikipedia.org/wiki/WebSocket)

The following example disables the HTTP long-polling transport:

```
IO.Options options = IO.Options.builder()
        .setTransports(new String[] { WebSocket.NAME })
        .build();

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

Note: in that case, sticky sessions are not required on the server side (more information here (https://socket.io/docs/v4/using-multiple-nodes/) ).

`upgrade`

Default value: `true`

Whether the client should try to upgrade the transport from HTTP long-polling to something better.

`rememberUpgrade`

Default value: `false`

If true and if the previous WebSocket connection to the server succeeded, the connection attempt will bypass the normal upgrade process and will initially try WebSocket. A connection attempt following a transport error will use the normal upgrade process. It is recommended you turn this on only when using SSL/TLS connections, or if you know that your network does not block websockets.

`path`

Default value: `/socket.io/`

It is the name of the path that is captured on the server side.

The server and the client values must match:

*Client*

```
IO.Options options = IO.Options.builder()
        .setPath("/my-custom-path/")
        .build();

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

*JavaScript Server*

```
import { Server } from "socket.io";

const io = new Server(8080, {
  path: "/my-custom-path/"
});

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

Please note that this is different from the path in the URI, which represents the Namespace (https://socket.io/docs/v4/namespaces/) .

Example:

```
IO.Options options = IO.Options.builder()
        .setPath("/my-custom-path/")
        .build();

Socket socket = IO.socket(URI.create("https://example.com/order"), options);
```

- the Socket instance is attached to the “order” Namespace
- the HTTP requests will look like: `GET https://example.com/my-custom-path/?EIO=4&transport=polling&t=ML4jUwU`

`query`

Default value: -

Additional query parameters (then found in `socket.handshake.query` object on the server-side).

Example:

*Client*

```
IO.Options options = IO.Options.builder()
        .setQuery("x=42")
        .build();

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

*JavaScript Server*

```
io.on("connection", (socket) => {
  console.log(socket.handshake.query); // prints { x: '42', EIO: '4', transport: 'polling' }
});
```

Note: The `socket.handshake.query` object contains the query parameters that were sent during the Socket.IO handshake, it won’t be updated for the duration of the current session, which means changing the `query` on the client-side will only be effective when the current session is closed and a new one is created:

```
socket.io().on(Manager.EVENT_RECONNECT_ATTEMPT, new Emitter.Listener() {
    @Override
    public void call(Object... args) {
        options.query = "y=43";
    }
});
```

`extraHeaders`

Default value: -

Additional headers (then found in `socket.handshake.headers` object on the server-side).

Example:

*Client*

```
IO.Options options = IO.Options.builder()
        .setExtraHeaders(singletonMap("authorization", singletonList("bearer 1234")))
        .build();

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

*JavaScript Server*

```
io.on("connection", (socket) => {
  console.log(socket.handshake.headers); // prints { accept: '*/*', authorization: 'bearer 1234', connection: 'Keep-Alive', 'accept-encoding': 'gzip', 'user-agent': 'okhttp/3.12.12' }
});
```

Note: Similar to the `query` option above, the `socket.handshake.headers` object contains the headers that were sent during the Socket.IO handshake, it won't be updated for the duration of the current session, which means changing the `extraHeaders` on the client-side will only be effective when the current session is closed and a new one is created:

```
socket.io().on(Manager.EVENT_RECONNECT_ATTEMPT, new Emitter.Listener() {
    @Override
    public void call(Object... args) {
        options.extraHeaders.put("authorization", singletonList("bearer 5678"));
    }
});
```

`callFactory`

The OkHttpClient instance (https://square.github.io/okhttp/4.x/okhttp/okhttp3/-ok-http-client/) to use for HTTP long-polling requests.

```
OkHttpClient okHttpClient = new OkHttpClient.Builder()
        .readTimeout(1, TimeUnit.MINUTES) // important for HTTP long-polling
        .build();

IO.Options options = new IO.Options();
options.callFactory = okHttpClient;

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

`webSocketFactory`

The OkHttpClient instance (https://square.github.io/okhttp/4.x/okhttp/okhttp3/-ok-http-client/) to use for WebSocket connections.

```
OkHttpClient okHttpClient = new OkHttpClient.Builder()
        .minWebSocketMessageToCompress(2048)
        .build();

IO.Options options = new IO.Options();
options.webSocketFactory = okHttpClient;

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

## Manager options

These settings will be shared by all Socket instances attached to the same Manager.

`reconnection`

Default value: `true`

Whether reconnection is enabled or not. If set to `false`, you need to manually reconnect.

`reconnectionAttempts`

Default value: `Integer.MAX_VALUE`

The number of reconnection attempts before giving up.

`reconnectionDelay`

Default value: `1_000`

The initial delay before reconnection in milliseconds (affected by the randomizationFactor value).

`reconnectionDelayMax`

Default value: `5_000`

The maximum delay between two reconnection attempts. Each attempt increases the reconnection delay by 2x.

`randomizationFactor`

Default value: `0.5`

The randomization factor used when reconnecting (so that the clients do not reconnect at the exact same time after a server crash, for example).

Example with the default values:

- 1st reconnection attempt happens between 500 and 1500 ms (`1000 * 2^0 * (<something between -0.5 and 1.5>)`)
- 2nd reconnection attempt happens between 1000 and 3000 ms (`1000 * 2^1 * (<something between -0.5 and 1.5>)`)
- 3rd reconnection attempt happens between 2000 and 5000 ms (`1000 * 2^2 * (<something between -0.5 and 1.5>)`)
- next reconnection attempts happen after 5000 ms

`timeout`

Default value: `20_000`

The timeout in milliseconds for each connection attempt.

## Socket options

These settings are specific to the given Socket instance.

`auth`

Default value: -

Credentials that are sent when accessing a namespace (see also here (https://socket.io/docs/v4/middlewares/#sending-credentials) ).

Example:

*Client*

```
IO.Options options = IO.Options.builder()
        .setAuth(singletonMap("token", "abcd"))
        .build();

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

*JavaScript Server*

```
io.on("connection", (socket) => {
  console.log(socket.handshake.auth); // prints { token: 'abcd' }
});
```

You can update the `auth` map when the access to the Namespace is denied:

```
socket.on(Socket.EVENT_CONNECT_ERROR, new Emitter.Listener() {
    @Override
    public void call(Object... args) {
        options.auth.put("token", "efgh");
        socket.connect();
    }
});
```

Or manually force the Socket instance to reconnect:

```
options.auth.put("token", "efgh");
socket.disconnect().connect();
```

# SSL connections

## With a keystore

```
HostnameVerifier hostnameVerifier = new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession sslSession) {
        return hostname.equals("example.com");
    }
};

KeyStore ks = KeyStore.getInstance("JKS");
File file = new File("path/to/the/keystore.jks");
ks.load(new FileInputStream(file), "password".toCharArray());

KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
kmf.init(ks, "password".toCharArray());

TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(ks);

SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

OkHttpClient okHttpClient = new OkHttpClient.Builder()
        .hostnameVerifier(hostnameVerifier)
        .sslSocketFactory(sslContext.getSocketFactory(), (X509TrustManager) tmf.getTrustManagers()[0])
        .readTimeout(1, TimeUnit.MINUTES) // important for HTTP long-polling
        .build();

IO.Options options = new IO.Options();
options.callFactory = okHttpClient;
options.webSocketFactory = okHttpClient;

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

## Trust all certificates

Please use with caution, as this defeats the whole purpose of using secure connections.

This is equivalent to `rejectUnauthorized: false` for the JavaScript client.

```java
HostnameVerifier hostnameVerifier = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession sslSession) {
        return true;
    }
};

X509TrustManager trustManager = new X509TrustManager() {
    public X509Certificate[] getAcceptedIssuers() {
        return new X509Certificate[] {};
    }

    @Override
    public void checkClientTrusted(X509Certificate[] arg0, String arg1) {
        // not implemented
    }

    @Override
    public void checkServerTrusted(X509Certificate[] arg0, String arg1) {
        // not implemented
    }
};

SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, new TrustManager[] { trustManager }, null);

OkHttpClient okHttpClient = new OkHttpClient.Builder()
        .hostnameVerifier(hostnameVerifier)
        .sslSocketFactory(sslContext.getSocketFactory(), trustManager)
        .readTimeout(1, TimeUnit.MINUTES) // important for HTTP long-polling
        .build();

IO.Options options = new IO.Options();
options.callFactory = okHttpClient;
options.webSocketFactory = okHttpClient;

Socket socket = IO.socket(URI.create("https://example.com"), options);
```

## Multiplexing

The Java client does support multiplexing: this allows to split the logic of your application into distinct modules, while using one single WebSocket connection to the server.

Reference: https://socket.io/docs/v4/namespaces/ (https://socket.io/docs/v4/namespaces/)

```
Socket socket = IO.socket(URI.create("https://example.com")); // the main namespace
Socket productSocket = IO.socket(URI.create("https://example.com/product")); // the "product" namespac
e
Socket orderSocket = IO.socket(URI.create("https://example.com/order")); // the "order" namespace

// all 3 sockets share the same Manager
System.out.println(socket.io() == productSocket.io()); // true
System.out.println(socket.io() == orderSocket.io()); // true
```

Please note that multiplexing will be disabled in the following cases:

- multiple creation for the same namespace

```
Socket socket = IO.socket(URI.create("https://example.com"));
Socket socket2 = IO.socket(URI.create("https://example.com"));

System.out.println(socket.io() == socket2.io()); // false
```

- different domains

```
Socket socket = IO.socket(URI.create("https://first.example.com"));
Socket socket2 = IO.socket(URI.create("https://second.example.com"));

System.out.println(socket.io() == socket2.io()); // false
```

- usage of the forceNew option

```
IO.Options options = IO.Options.builder()
    .setForceNew(true)
    .build();

Socket socket = IO.socket(URI.create("https://example.com"));
Socket socket2 = IO.socket(URI.create("https://example.com/admin"), options);

System.out.println(socket.io() == socket2.io()); // false
```