Socket.IO - Quick Guide

Previous Page

Next Page

Overview

Socket.IO

Socket.IO is a JavaScript library for realtime web applications. It enables realtime, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library for node.js. Both components have a nearly identical API.

Realtime applications

A real-time application (RTA) is an application that functions within a time frame that the user senses as immediate or current. Some examples of real-time applications are:

- Instant messengers: Chat apps like Whatsapp, Facebook messenger, etc. You need not refresh your app/website to receive new messages.
- Push Notifications: When someone tags you in a picture on facebook, you receive an notification instantly.
- Collaboration applications: Apps like google docs, which allow multiple people to update same documents simultaneously and apply changes to all people's instances.
- Online gaming: Games like Counter Strike, Call of Duty, etc are also some examples of real time applications.

Why Socket.IO

Writing a realtime applications with popular web applications stacks like LAMP (PHP) has traditionally been very hard. It involves polling the server for changes, keeping track of timestamps, and it's a lot slower than it should be.

Sockets have traditionally been the solution around which most realtime systems are architected, providing a bi-directional communication channel between a client and a server. This means that the server can push messages to clients. Whenever an event occurs, the idea is that the server will get it and push it to concerned connected clients.

Socket.IO is quite popular, it is used by Microsoft Office, Yammer, Zendesk, Trello, and numerous other organizations to build robust real time systems. It one of the most powerful JavaScript frameworks on GitHub, and most depended-upon NPM module. Socket.IO also has a huge community, which means finding help is quite easy.

ExpressJS

We'll be using express to build the web server that Socket.IO will work with. Any other node server side framework or even node HTTP server can be used. But express makes it easy to define routes and other things. To read more about express and get a basic idea about it, head to: ExpressJS tutorial .

Environment

To get started with developing using the **Socket.IO**, you need to have **Node** and **npm(node package manager)** installed. If you don't already have these, head over to Node setup to install node on your local system. Confirm that node and npm are installed by running the following commands in your terminal.

```
node --version
npm --version
```

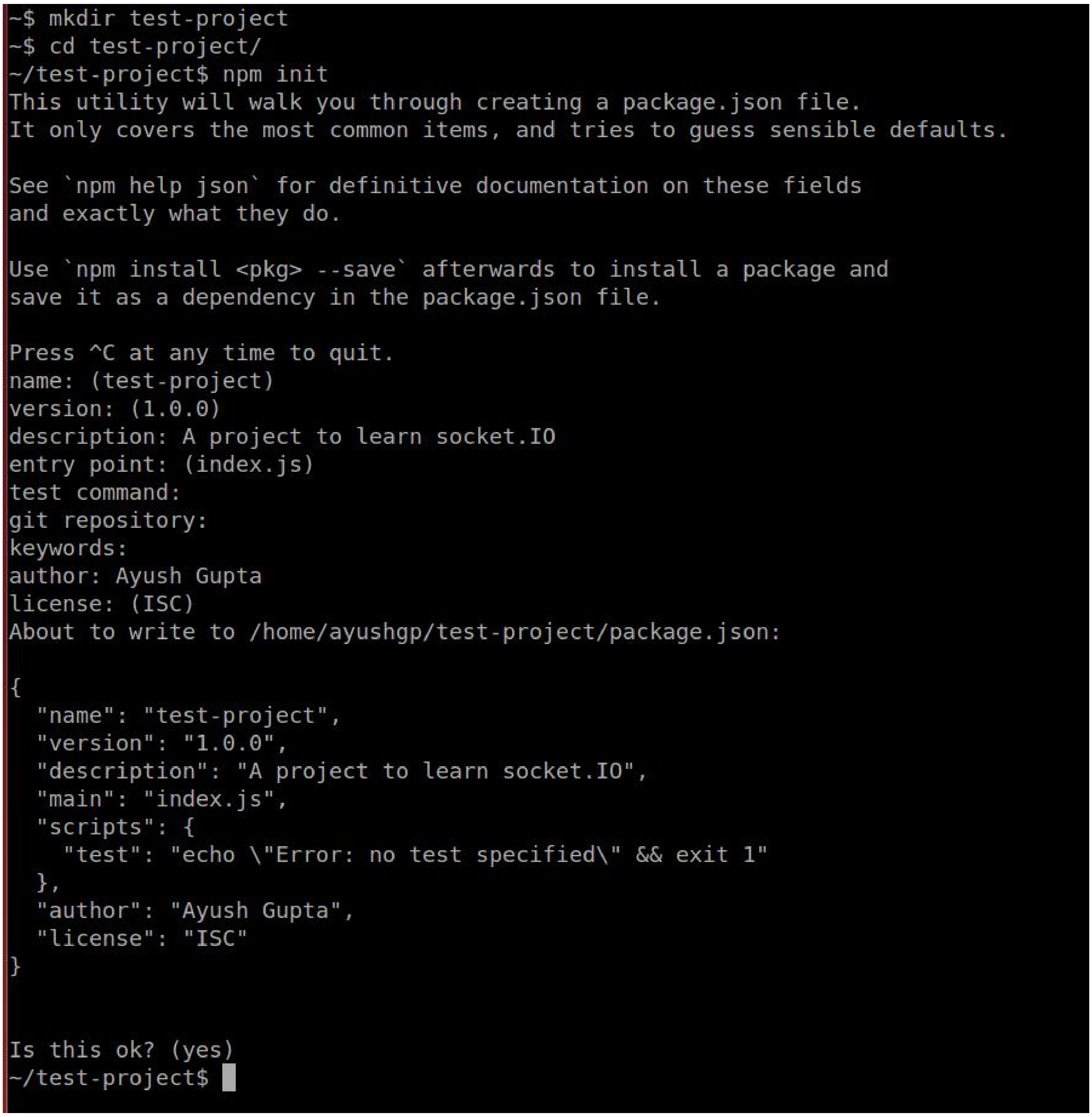
You should get an output similar to:

```
v5.0.0
3.5.2
```

Open your terminal and enter the following in your terminal to create a new folder and enter the following commands:

```
$ mkdir test-project
$ cd test-proect
$ npm init
```

It'll ask you some questions, answer them in the following way:



```
~$ mkdir test-project
~$ cd test-project/
~/test-project$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (test-project)
version: (1.0.0)
description: A project to learn socket.IO
entry point: (index.js)
test command:
git repository:
keywords:
author: Ayush Gupta
license: (ISC)
About to write to /home/ayushgp/test-project/package.json:

{
  "name": "test-project",
  "version": "1.0.0",
  "description": "A project to learn socket.IO",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ayush Gupta",
  "license": "ISC"
}


Is this ok? (yes)
~/test-project$ 
```

This will create a package.json node.js configuration file. Now we need to install **express** and **Socket.IO**. To install these and save them to package.json file, enter the following command in your terminal, in your project directory:

```
npm install --save express socket.io
```

One final thing we need to keep restarting the server when we make changes is a tool called nodemon. To install nodemon open your terminal and enter:

```
npm install -g nodemon
```

Whenever you need to start the server, instead of using *node app.js* use, *nodemon app.js*. This will ensure that you dont need to restart the server whenevr you change a file. It speeds up our development process.

Now we have our development environment set up. Lets get started with developing real time applications with socket.IO.

Hello World

Create a file called app.js and enter the following to set up an express application:

```
var app = require('express')();
var http = require('http').Server(app);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

We'll need an index.html file to serve, create a new file called index.html and enter the following in it:

```
<!DOCTYPE html>
<html>
  <head><title>Hello world</title></head>
  <body>Hello world</body>
</html>
```

To test if this works, go to your terminal and run this app using

```
nodemon app.js
```

This will run your server on localhost:3000. Go to your browser and enter localhost:3000 to check this.

This set up our express application and is now serving a HTML file on the root route. Now we will require socket.IO and will log "A user connected", everytime a user goes to this page and "A user disconnected", everytime someone navigates away/closes this page.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

//Whenever someone connects this gets executed
io.on('connection', function(socket){
  console.log('A user connected');

  //Whenever someone disconnects this piece of code executed
  socket.on('disconnect', function () {
    console.log('A user disconnected');
  });

});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

The require('socket.io')(http) creates a new socket.io instance attached to the http server. The io.on event handler handles connection, disconnection, etc events in it, using the socket object.

We have set uo our server to log messages on connections and disconnections. Now also need to include the client script and initialize the socket object there so that clients can establish connections when required. The script is served by our io server at **'/socket.io/socket.io.js'**. After doing the above, the index.html file will look like:

```
<!DOCTYPE html>
<html>
  <head><title>Hello world</title></head>
  <script src="/socket.io/socket.io.js"></script>
  <script>
    var socket = io();
  </script>
  <body>Hello world</body>
</html>
```

If you go to localhost:3000 now(make sure your server is running), you'll get Hello world printed in your browser. Now check your server console logs, it'll show a message:

```
A user connected
```

If you refresh your browser, it'll disconnect the socket connection and recreate. You can see this on your console logs:

```
A user connected
A user disconnected
A user connected
```

We now have socket connections working. This is how easy it is to set up connections in socket.IO.

Event Handling

Sockets work based on events. There are some reserved events that can be accessed using the socket object on the server side: connect, message, disconnect, reconnect, ping, join and leave. The client side socket object also provides us with some reserved events: connect, connect_error, connect_timeout, reconnect, etc.

In the hello world example we used the connection and disconnect events to log when a user connected and left. Now we'll be using the message event to pass message from the server to the client. To do this, modify the io.on('connection', function(socket)) call to include the following:

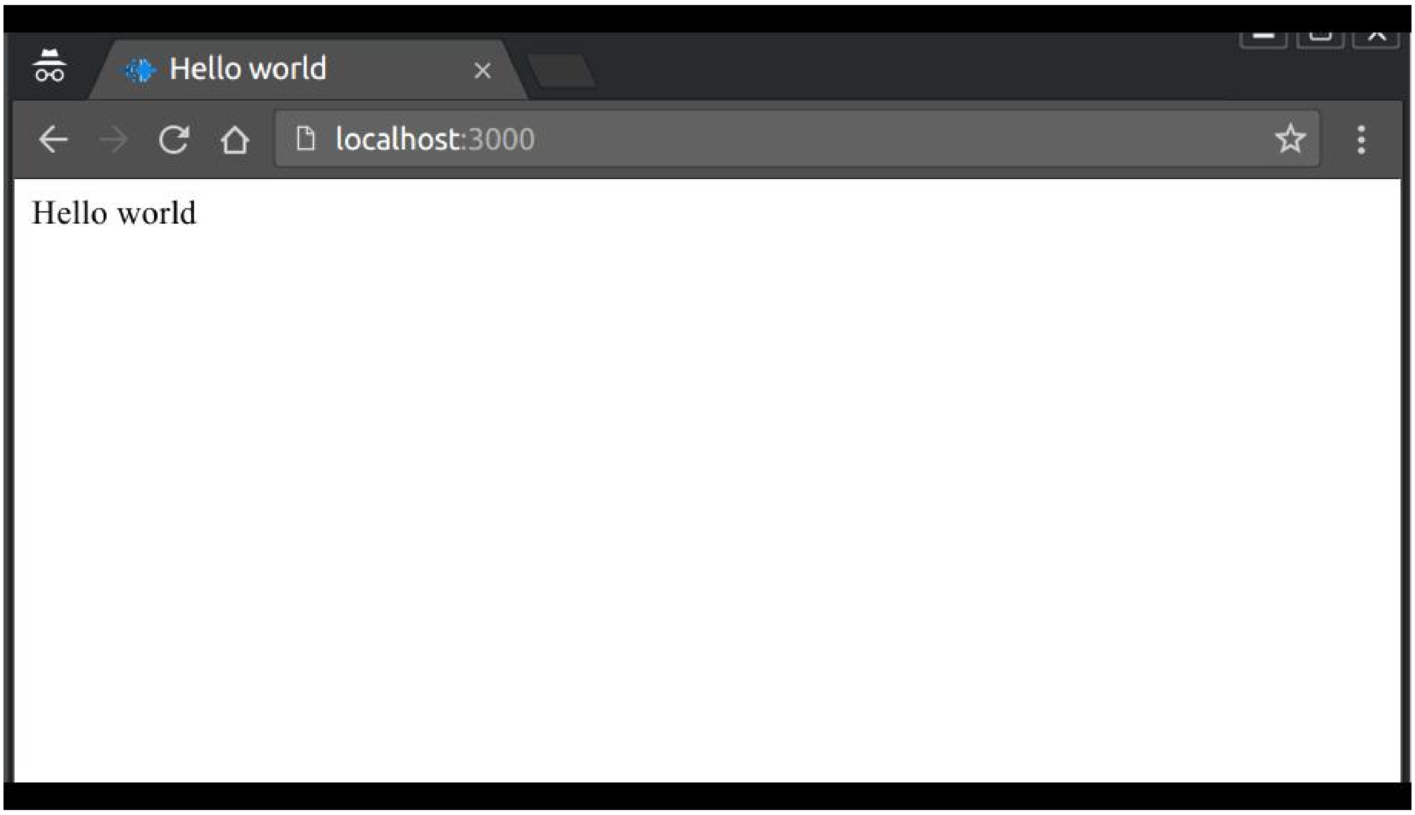
```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

io.on('connection', function(socket){
  console.log('A user connected');

  //Send a message after a timeout of 4seconds
  setTimeout(function(){
    socket.send('Sent a message 4seconds after connection!');
  }, 4000);
  socket.on('disconnect', function () {
    console.log('A user disconnected');
  });
});
http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

This will send an event called message(built in) to our client, 4 seconds after the client connects. The send function on socket object associates the 'message' event.

Now we need to handle this event on our client side. So edit your index.html script tag to include the following code:

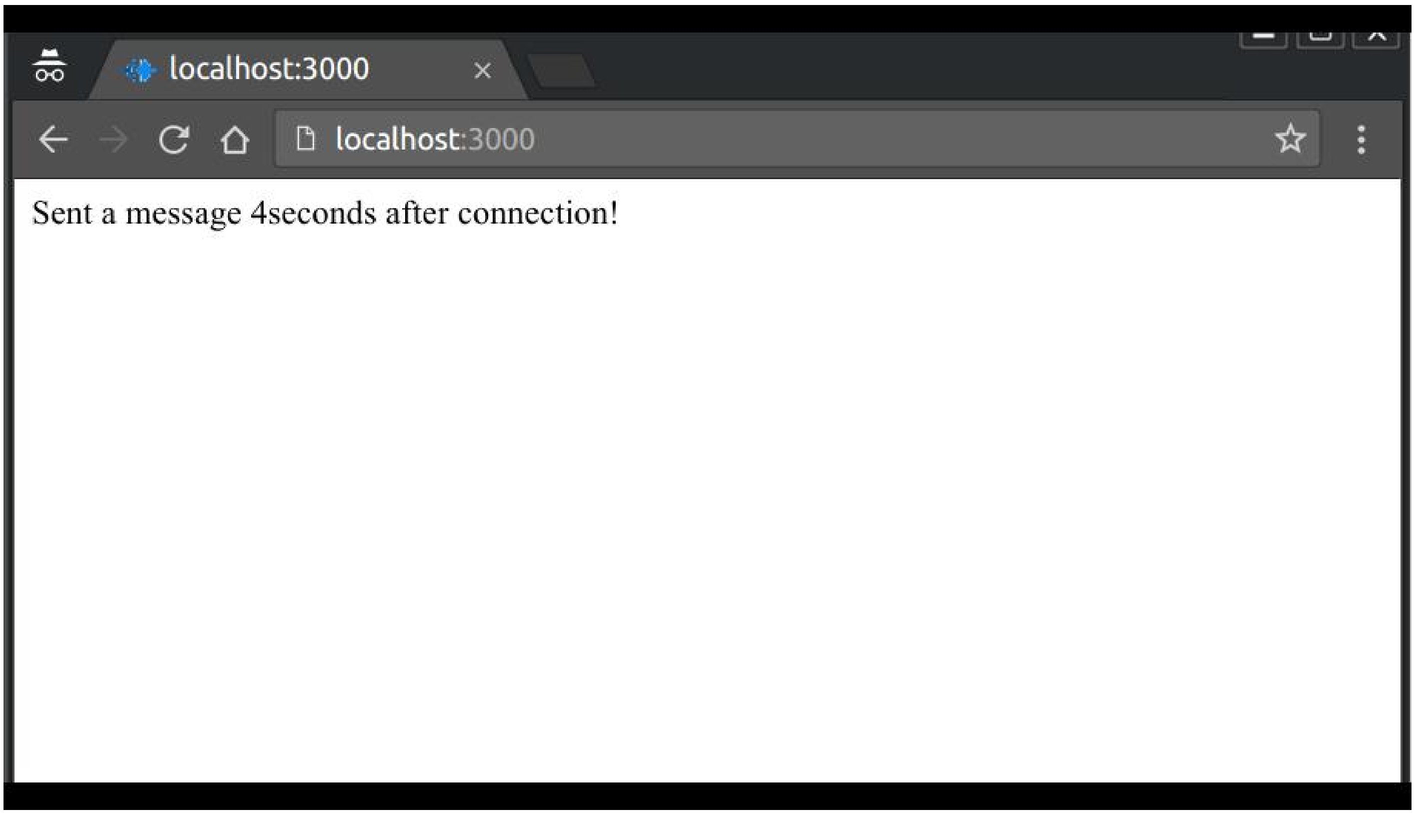
```
<script>
    var socket = io();
    socket.on('message', function(data){document.write(data)});
</script>
```

Now we are handling the 'message' event on the client. When you go to the page in your browser now, you'll be presented with:



Hello world

After 4 seconds pass and the server sends the message event, our client will handle it and produce the following output:



Sent a message 4seconds after connection!

Note that we sent just a string of text here, we can also send an object in any event.

Message was a built in event provided by the API but is of not much use in a real application as we need to be able to differentiate between events. To allow this, socket.IO provide us the ability to create custom events. You can create and fire custom events using the **socket.emit** function. For example, the below code emits an event called *testerEvent*:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

io.on('connection', function(socket){
  console.log('A user connected');
  //Send a message when
  setTimeout(function(){
   //Sending an object when emmiting an event
 socket.emit('testerEvent', { description: 'A custom event named testerEvent!'});
 }, 4000);
  socket.on('disconnect', function () {
    console.log('A user disconnected');
  });
});

http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

To handle this custom event on client we need a listener that listens for the event testerEvent. The following code handles this event on the client:

```
<!DOCTYPE html>
<html>
 <head><title>Hello world</title></head>
 <script src="/socket.io/socket.io.js"></script>
 <script>
   var socket = io();
   socket.on('testerEvent', function(data){document.write(data.description)});
 </script>
 <body>Hello world</body>
</html>
```

This will work in the same way as our previous example, with the event being testerEvent in this case. When you open your browser and go to localhost:3000, you'll be greeted with:

```
Hello world
```

After 4 seconds, this event will be fired and your browser will have the text changed to:

```
A custom event named testerEvent!
```

We can also emit events from the client. To emit an event from your client, use the emit function on the socket object.

```
<!DOCTYPE html>
<html>
 <head><title>Hello world</title></head>
 <script src="/socket.io/socket.io.js"></script>
 <script>
   var socket = io();
   socket.emit('clientEvent', 'Sent an event from the client!');
 </script>
 <body>Hello world</body>
</html>
```

And to handle these events, use the on function on the socket object on your server.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

io.on('connection', function(socket){
  socket.on('clientEvent', function(data){
 console.log(data);
  });
});

http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

So now if you go to localhost:3000, you'll get a custom event called clientEvent fired. This event will be handled on the server by logging:

```
Sent an event from the client!
```

Broadcasting

Broadcasting means sending a message to all connected clients. Broadcasting can be done at multiple levels. We can send the message to all connected clients, to clients on a namespace and clients in a particular room. The latter 2 would be covered in their respective chapters.

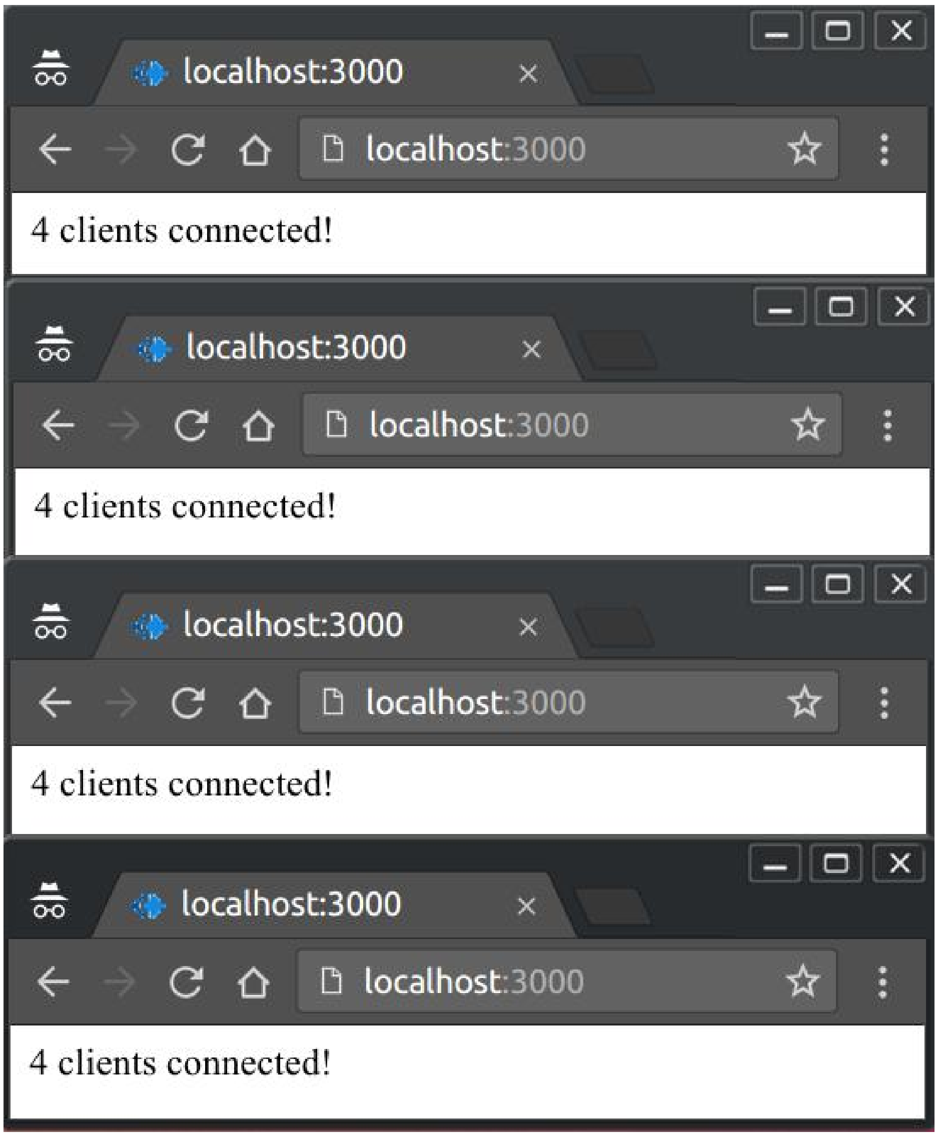
To broadcast an event to all the clients, We can use the **io.sockets.emit** method. Note that this will emit the event to ***ALL*** the connected clients(event the socket that might have fired this event). In this example we will broadcast the number of connected clients to all the users. Update your app.js file to incorporate the folllowing:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

var clients = 0;

io.on('connection', function(socket){
  clients++;
 io.sockets.emit('broadcast',{ description: clients + ' clients connected!'});
 socket.on('disconnect', function () {
        clients--;
          io.sockets.emit('broadcast',{ description: clients + ' clients connected!'});
             });
});

http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

On the client side, we just need to handle the broadcast event:

```
<!DOCTYPE html>
<html>
 <head><title>Hello world</title></head>
 <script src="/socket.io/socket.io.js"></script>
 <script>
   var socket = io();
   socket.on('broadcast',function(data){
     document.body.innerHTML = '';
     document.write(data.description);
   });
 </script>
 <body>Hello world</body>
</html>
```

If you connect 4 clients, you will get the following result:



4 clients connected!

4 clients connected!

4 clients connected!

4 clients connected!

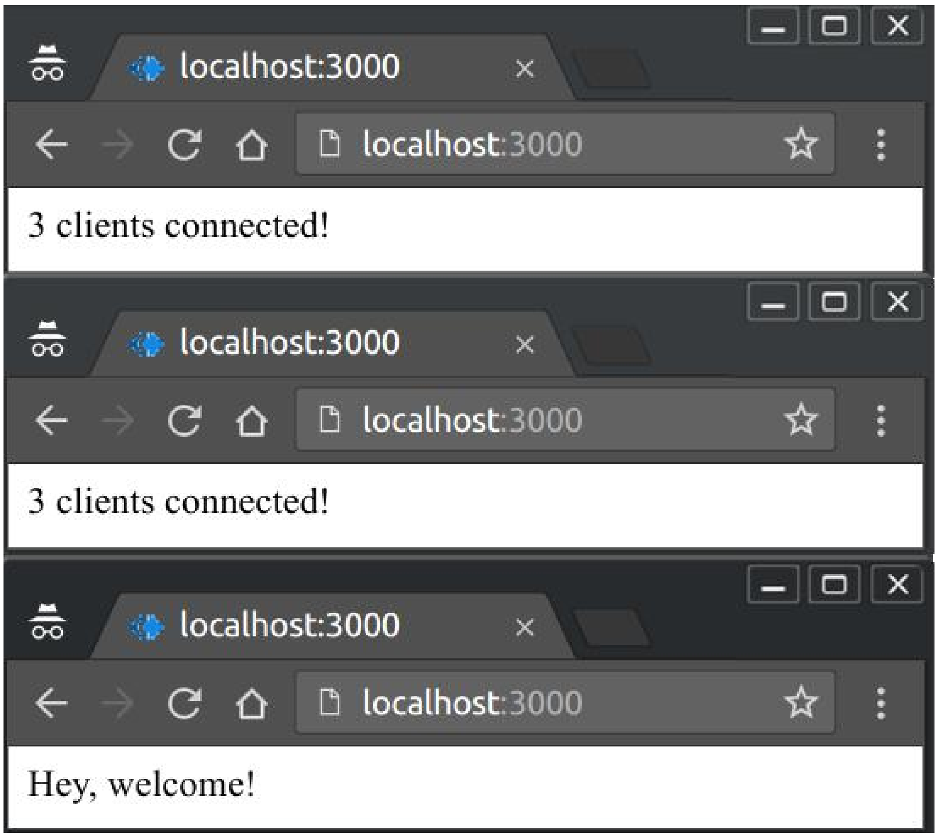
This was to send an event to everyone. Now if we want to send an event to everyone but the client that caused(in previous example it was caused by new clients on connecting), we can use the socket.broadcast.emit. Lets send the new user a welcome message and update the other clients about him/her joining. So in your app.js file, on connection of client send him a welcome message and broadcast connected client number to all others.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

var clients = 0;

io.on('connection', function(socket){
  clients++;
socket.emit('newclientconnect',{ description: 'Hey, welcome!'});
socket.broadcast.emit('newclientconnect',{ description: clients + ' clients connected!'})
 socket.on('disconnect', function () {
        clients--;
  socket.broadcast.emit('newclientconnect',{ description: clients + ' clients connected!'})
            });
});

http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

And your html to handle this event:

```
<!DOCTYPE html>
<html>
 <head><title>Hello world</title></head>
 <script src="/socket.io/socket.io.js"></script>
 <script>
   var socket = io();
   socket.on('newclientconnect',function(data){
     document.body.innerHTML = '';
     document.write(data.description);
  });
 </script>
 <body>Hello world</body>
</html>
```

Now the newest client gets a welcome message and others get how many clients are connected currently to the server.



3 clients connected!

3 clients connected!

Hey, welcome!

Namespaces

Socket.IO allows you to “namespace” your sockets, which essentially means assigning different endpoints or paths.

This is a useful feature to minimize the number of resources (TCP connections) and at the same time separate concerns within your application by introducing separation between communication channels. Multiple namespaces actually share the same WebSockets connection thus saving us socket ports on the server.

Namespaces are created on the server side. But they are joined by clients by sending a request to the server.

Default Namespaces

The root namespace '/' is the default namespace which is joined by clients if a namespace is not specified by the client while connecting to the server. All connections to the server using the socket object client side are made to the default namespace. For example,

```
var socket = io();
```

This will connect the client to the default namespace. All events on this namespace connection will be handled by the io object on the server. All the previous examples were utilizing default namespaces to communicate with the server and back.

Custom Namespaces

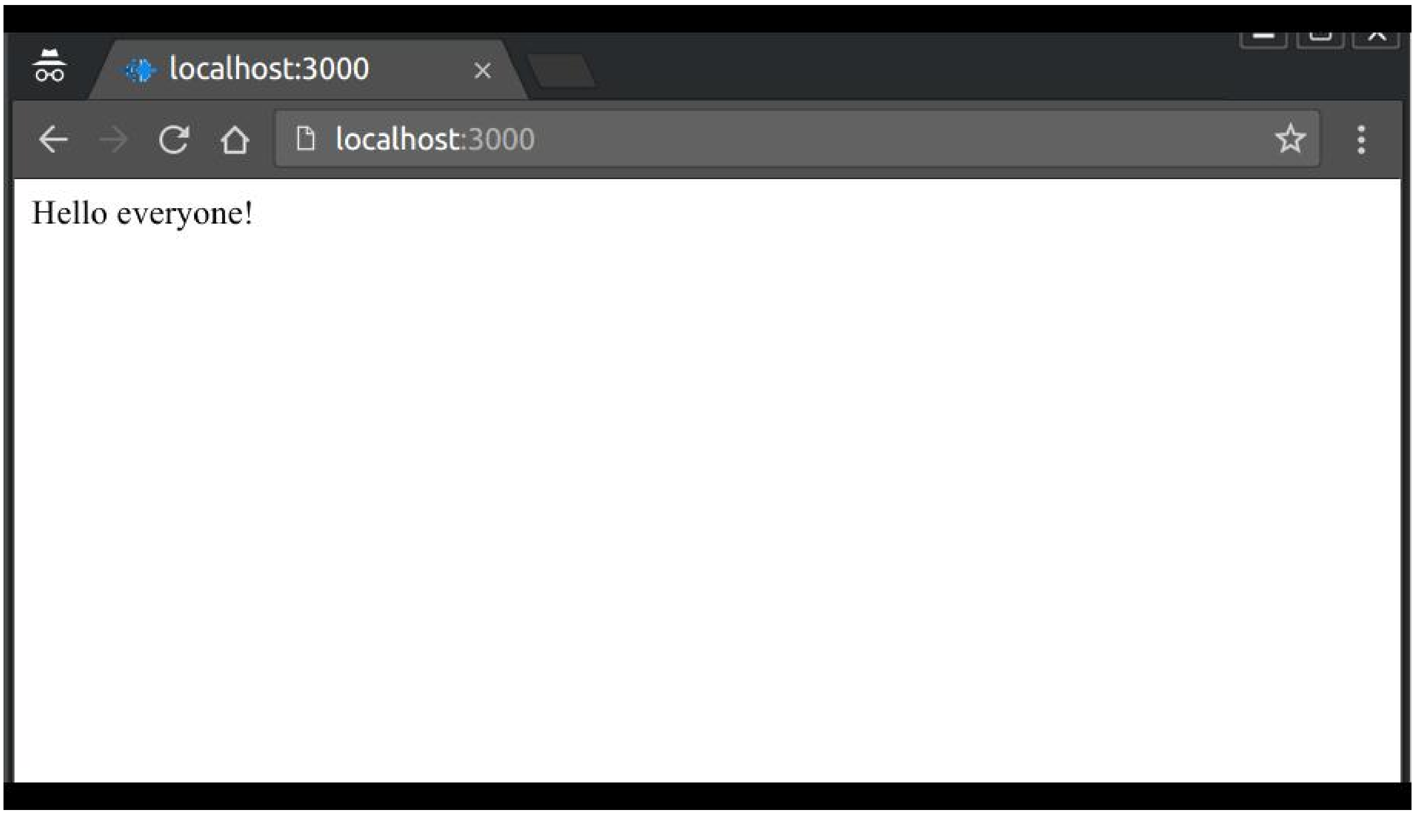
We can create our own custom namespaces. To set up a custom namespace, we can call the of function on the server-side:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});

var nsp = io.of('/my-namespace');
nsp.on('connection', function(socket){
  console.log('someone connected');
  nsp.emit('hi', 'Hello everyone!');
});
http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

Now to connect a client to this namespace, you need to provide the namespace as an argument to the io constructor call to create a connection and a socket object on client side. For example, to connect to the above namespace, use the following HTML:

```
<!DOCTYPE html>
<html>
 <head><title>Hello world</title></head>
 <script src="/socket.io/socket.io.js"></script>
 <script>
   var socket = io('/my-namespace');
   socket.on('hi',function(data){
     document.body.innerHTML = '';
     document.write(data);
   });
 </script>
 <body></body>
</html>
```

Everytime someone connects to this namespace, they'll receive a hi event.



Hello everyone!

Rooms

Within each namespace, you can also define arbitrary channels that sockets can join and leave. These channels are called rooms. Rooms are used to further separate concerns.

Rooms also share the same socket connection like namespaces.

One thing to keep in mind while using rooms is that they can only be joined on the server side.

Joining rooms

You can call the **join** method on the socket to subscribe the socket to a given channel/room. For example, lets create rooms called **'room-<room-number>'** and join some clients. As soon as this room is full, create another room and join clients there. Note that we are currently doing this on the default namespace, ie, '/'. You can also implement this in custom namespaces in the same fashion.

To join a room you need to provide the room name as the argument to your join function call.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});
var roomno = 1;
io.on('connection', function(socket){
  //Increase roomno 2 clients are present in a room.
  if(io.nsps['/'].adapter.rooms["room-"+roomno] && io.nsps['/'].adapter.rooms["room-"+roomno].length > 1)
    roomno++;
  socket.join("room-"+roomno);

  //Send this event to everyone in the room.
  io.sockets.in("room-"+roomno).emit('connectToRoom', "You are in room no. "+roomno);
})
http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

Just handle this connectToRoom event on the client.

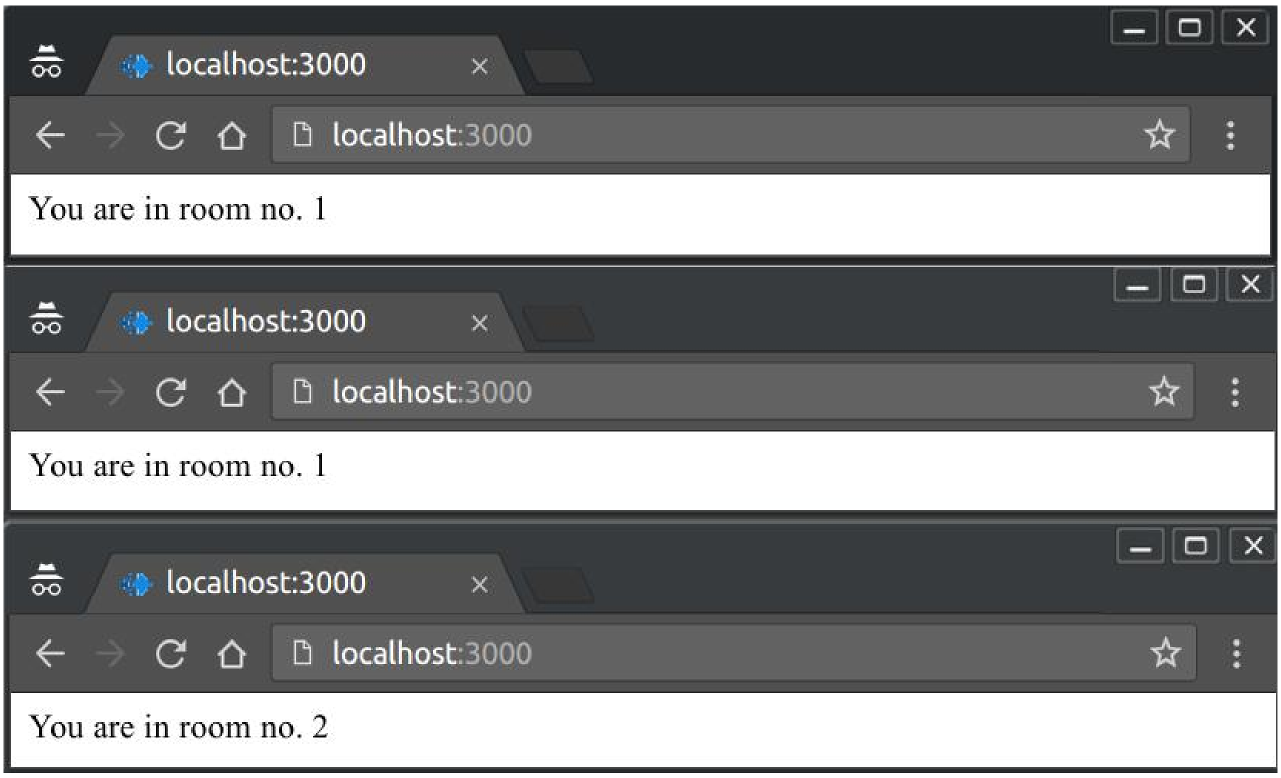
```
<!DOCTYPE html>
<html>
    <head><title>Hello world</title></head>
    <script src="/socket.io/socket.io.js"></script>
    <script>
      var socket = io();
      socket.on('connectToRoom',function(data){
          document.body.innerHTML = '';
          document.write(data);
      });
    </script>
    <body></body>
</html>
```

Now if you connect 3 clients, first 2 will get the message:

```
You are in room no. 1
```

Next one will get the message:

```
You are in room no. 2
```



You are in room no. 1

You are in room no. 1

You are in room no. 2

Leaving a room

To leave a room, you need to call the leave function just like you called the join function on the socket. For example, to leave room **'room-1'**,

```
socket.leave("room-"+roomno);
```

Error Handling

We've worked on local servers till now which will almost never give us errors related to connections, timeouts, etc. But in real life production environments, handling such errors are of utmost importance. So we'll now discuss how we can handle connection errors on the client side.

The client API provides us with following built in events:

- ***connect***
  When the client successfully connects.
- ***connecting***
  When the client is in process of connecting.
- ***disconnect***
  When the client is disconnected
- ***connect_failed***
  When connection to server fails
- ***error***
  An error event is sent from the server
- ***message***
  When server sends a message using the **send** function.
- ***reconnect***
  When reconnection to server is successful.
- ***reconnecting***
  When the client is in process of connecting.
- ***reconnect_failed***
  When the reconnection attempt fails.

To handle errors, we can handle these events using out socket object that we created on our client. For example, if we have a connection that fails, we can use the following to connect to the server again:

```
socket.on('connect_failed', function() {
   document.write("Sorry, there seems to be an issue with the connection!");
})
```

Logging and Debugging

Socket.IO uses a very famous debugging module developed by ExpresJS's main author, called debug.

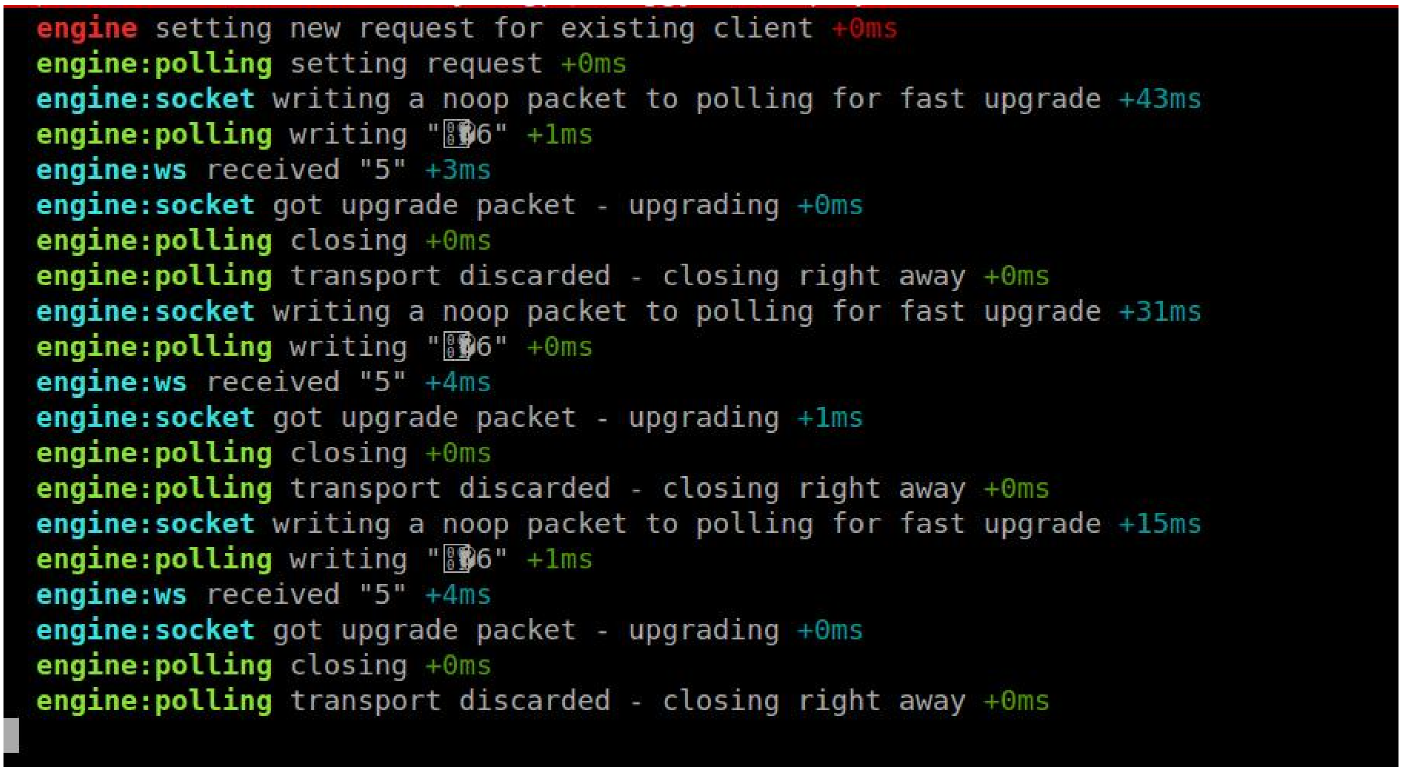
Earlier socket.IO used to log everything to the console making it quite difficult to actually debug the problem. After the v1.0 release, you can specify what you want to log.

Server-side

The best way to see what information is available is to use the *:

```
DEBUG=* node app.js
```

This will colorize and output everything that happens to your server console. For example,



```
engine setting new request for existing client +0ms
engine:polling setting request +0ms
engine:socket writing a noop packet to polling for fast upgrade +43ms
engine:polling writing "�6" +1ms
engine:ws received "5" +3ms
engine:socket got upgrade packet - upgrading +0ms
engine:polling closing +0ms
engine:polling transport discarded - closing right away +0ms
engine:socket writing a noop packet to polling for fast upgrade +31ms
engine:polling writing "�6" +0ms
engine:ws received "5" +4ms
engine:socket got upgrade packet - upgrading +1ms
engine:polling closing +0ms
engine:polling transport discarded - closing right away +0ms
engine:socket writing a noop packet to polling for fast upgrade +15ms
engine:polling writing "�6" +1ms
engine:ws received "5" +4ms
engine:socket got upgrade packet - upgrading +0ms
engine:polling closing +0ms
engine:polling transport discarded - closing right away +0ms
```

Client-side

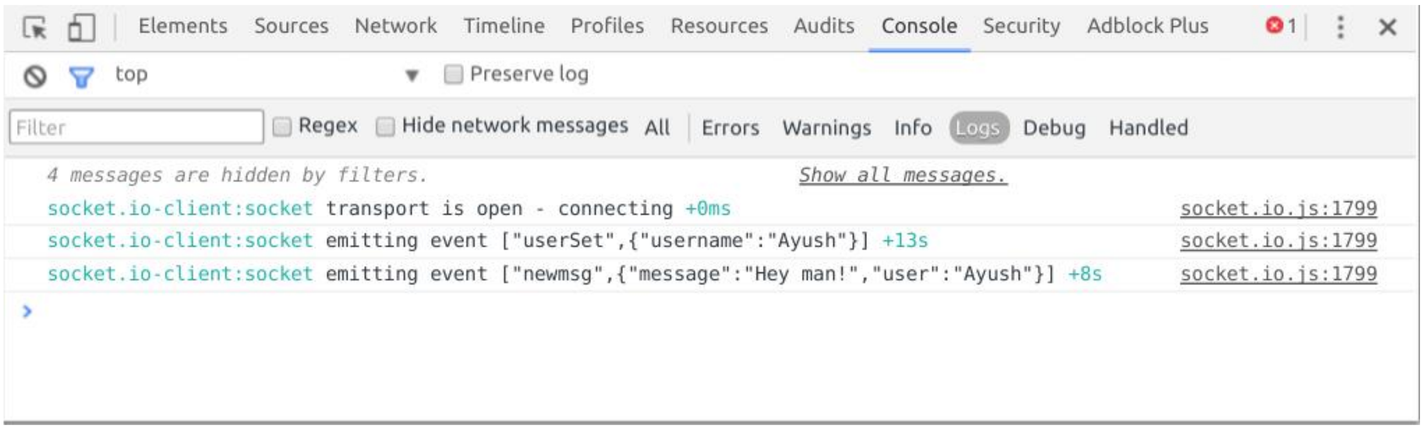
Paste this to console, click enter and refresh your page. This will again output everything related to socket.io to your console.

```
localStorage.debug = '*';
```

You can limit the output to only get debug info with incoming data from socket using:

```
localStorage.debug = 'socket.io-client:socket';
```

You can see result like the following if you use the second statement to log the info:



```
4 messages are hidden by filters.                                Show all messages.
socket.io-client:socket transport is open - connecting +0ms                                socket.io.js:1799
socket.io-client:socket emitting event ["userSet",{"username":"Ayush"}] +13s                socket.io.js:1799
socket.io-client:socket emitting event ["newmsg",{"message":"Hey man!","user":"Ayush"}] +8s socket.io.js:1799
```

There is a very good blog post related to socket.io debugging here.

Internals

Fallbacks

Socket.IO has a lot of underlying transport mechanisms, which deal with various constraints arising due to cross browser issues, WebSocket implementations, firewalls, port blocking, etc.

Though W3C has a defined specification for WebSocket API, it is still lacking in implementation. Socket.IO provides us with fallback mechanisms which can deal with such issues. If we develop apps using the native API, we have to implement the fallbacks ourselves. Socket.IO covers a large list of fallbacks in the following order:

- WebSockets
- FlashSocket
- XHR long polling
- XHR multipart streaming
- XHR polling
- JSONP polling
- iframes

Connection using Socket.IO

The socket.io connection begins with the handshake. This makes the handshake a special part of the protocol. Apart from the handshake, all the other events and messages in the protocol are transferred over the socket.

Socket.io is intended for use with web applications, and therefore it is assumed that these applications will always be able to use HTTP. It is because of this reasoning that the socket.io handshake takes place over HTTP using a POST request on the handshake URI(passed to the connect method).

Events and messages

WebSocket native API only sends messages across. Socket.IO provides an addition layer over these messages which allow us to create events and again helps us develop apps easily by seperating the different types of messages sent.

The native API sends messages only in plain text. This is also taken care of by socket.IO. It handles the serialization and deserialization of data for us.

We have an official client API for the web. For other clients such as native mobile phones, other application clients also we can use socket.IO using the following steps.

First, a connection needs to be established using the same connection protocol discussed above.

Second, the messages need to be in the same format as specified by socket.IO. This format enables socket.io to determine the type of the message and the data sent in the message, and some metadata useful for operation. The message format is:

```
[type] : [id ('+')] : [endpoint] (: [data]
```

- **type** is a single digit integer, specifying what type message it is.
- **id** is message ID, an incremental integet used for acknowledgements.
- **endpoint** is the socket endpoint that the message is intended to be delivered to.
- **data** is the associated data to be delivered to the socket. In case of messages, it is treated as plain text, for other events, it is treated as JSON.

Chat Application

Now that we are well aquainted with Socket.IO, let us write a chat application which we can use to chat on different chat rooms. We will allow users to choose a username and allow them to chat using them. So first let us set up our HTML file to request for a username:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
     var socket = io();
   </script>
   <body>
      <input type="text" name="name" value="" placeholder="Enter your name!">
      <button type="button" name="button">Let me chat!</button>
   </body>
</html>
```

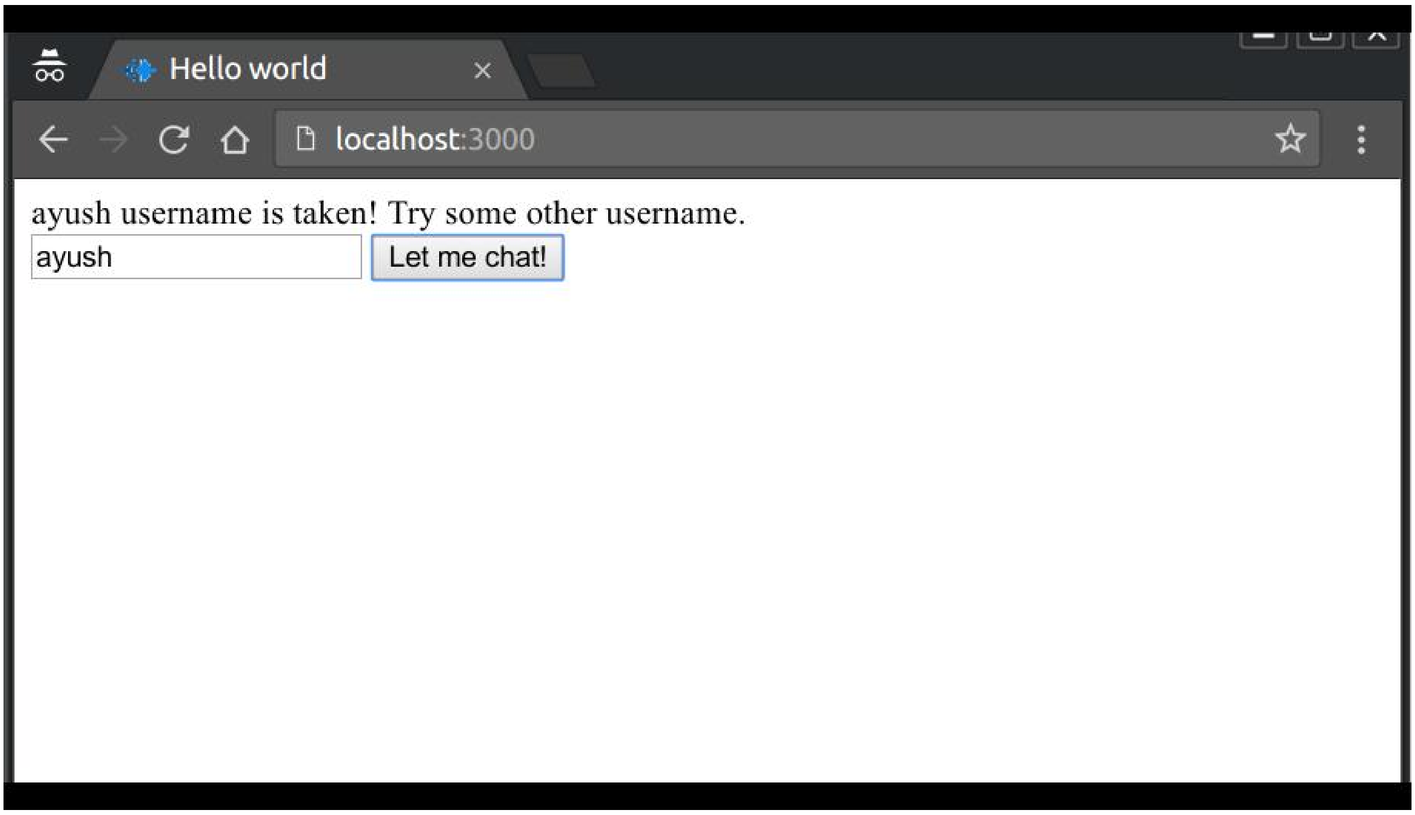
Now that we've set up our HTML to request for a username, let us create the server to accept connections from the client. We will allow people to send their chosen usernames using the **setUsername** event. If user exists, we'll respond by a **userExists** event, else using a **userSet** event.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});
users = [];
io.on('connection', function(socket){
  console.log('A user connected');
  socket.on('setUsername', function(data){
    if(users.indexOf(data) > -1){
      users.push(data);
      socket.emit('userSet', {username: data});
    }
    else{
      socket.emit('userExists', data + ' username is taken! Try some other username.');
    }
  })
});
http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

We need to send the username to the server when people click on the button. If user exists, we show an error message, else we show a messaging screen:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
    var socket = io();
      function setUsername(){
         socket.emit('setUsername', document.getElementById('name').value);
      };
      var user;
      socket.on('userExists', function(data){
         document.getElementById('error-container').innerHTML = data;
      });
      socket.on('userSet', function(data){
         user = data.username;
         document.body.innerHTML = '<input type="text" id="message">\
         <button type="button" name="button" onclick="sendMessage()">Send</button>\
         <div id="message-container"></div>';
      });
      function sendMessage(){
         var msg = document.getElementById('message').value;
         if(msg){
            socket.emit('msg', {message: msg, user: user});
         }
      }
      socket.on('newmsg', function(data){
         if(user){
            document.getElementById('message-container').innerHTML += '<div><b>' + data.user + '</b>: ' + data.message + '</div>'
         }
      })
   </script>
   <body>
      <div id="error-container"></div>
      <input id="name" type="text" name="name" value="" placeholder="Enter your name!">
      <button type="button" name="button" onclick="setUsername()">Let me chat!</button>
   </body>
</html>
```

Now if you connect 2 clients with same username, it'll give you an error like below:



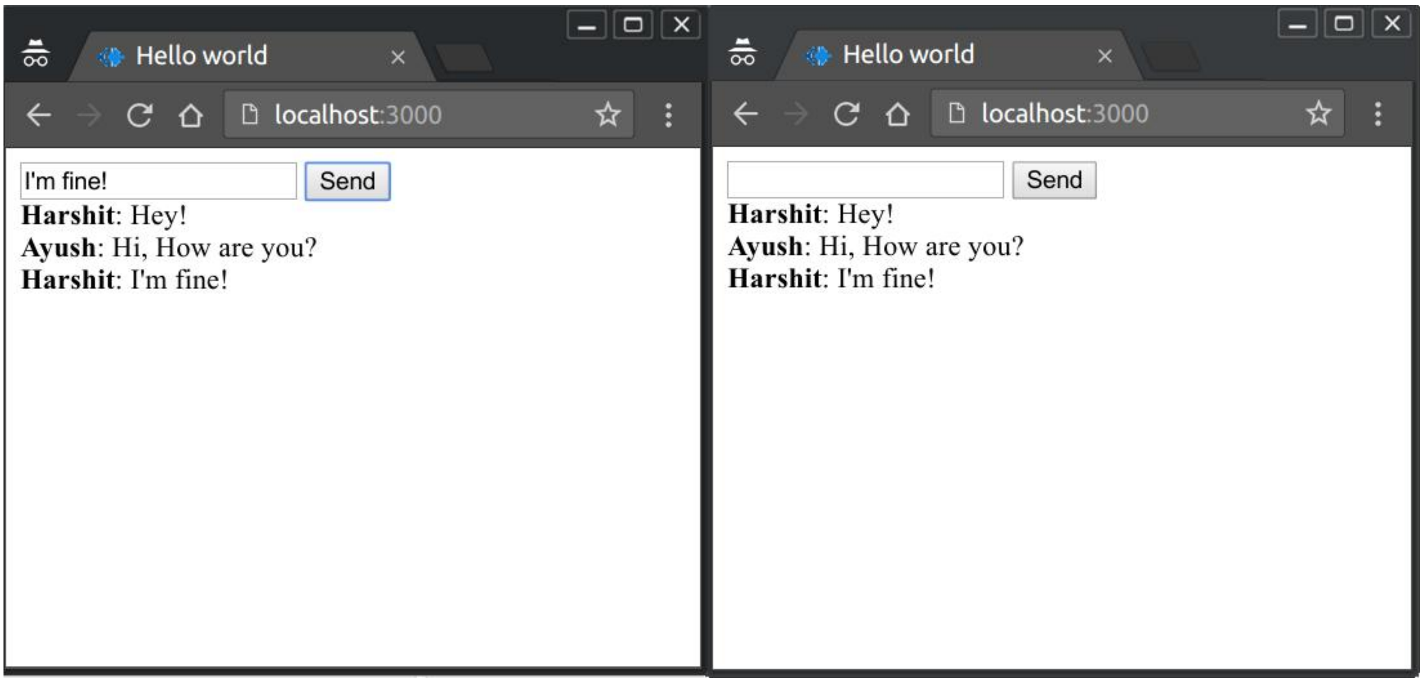
ayush username is taken! Try some other username.

Once you have provided an acceptable username, you'll be taken to a screen with a message box and a button to send messages.

Now we need to handle and direct the messages to connected client. For that modify your app.js file to include the following changes:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendfile('index.html');
});
users = [];
io.on('connection', function(socket){
  console.log('A user connected');
  socket.on('setUsername', function(data){
    console.log(data);
    if(users.indexOf(data) > -1){
      socket.emit('userExists', data + ' username is taken! Try some other username.');
    }
    else{
      users.push(data);
      socket.emit('userSet', {username: data});
    }
  });
  socket.on('msg', function(data){
      //Send message to everyone
      io.sockets.emit('newmsg', data);
  })
});
http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

Now connect any number of clients to your server, provide them a username and start chatting! In the following example I connnected 2 clients with names Ayush and Harshit and sent some messages from both the clients:



Harshit: Hey!
Ayush: Hi, How are you?
Harshit: I'm fine!

Harshit: Hey!
Ayush: Hi, How are you?
Harshit: I'm fine!

Previous Page

Next Page