CSCI 4140 – Tutorial 9

Building a chat room with Socket.IO in 20 minutes

Matt YIU, Man Tung (mtyiu@cse)
SHB 118
Office Hour: Wednesday, 3-5 pm

2016.03.29

Outline

- What is WebSocket?
- What is Socket.IO?
- Building a chat room with Socket.IO in 20 minutes
  - Lab 1: Display chat room UI
  - Lab 2: Integrate Socket.IO
  - Lab 3: Single chat room
  - Lab 4: Multiple chat rooms
- Namespaces and rooms in Socket.IO
- Socket.IO in Assignment 2

What is WebSocket?

- A protocol providing **full-duplex** (read & write) communications channels over a single **TCP connection**
- Designed to be implemented in **web browsers** and **web servers**
- A **dedicated server** is needed because an **application-level handshaking** is needed
- Other than that, WebSocket programming is the same as ordinary **socket programming**
- URI scheme: **`ws:`** and **`wss:`** for unencrypted and encrypted connections respectively (just like `http:` and `https:`)

What is Socket.IO?

- A JavaScript library for **realtime** web applications
- It enables **real-time bidirectional event-based** communications
- It primarily uses the **WebSocket** protocol with **polling** as a **fallback option**
  - It provides many more features than WebSocket, e.g., **broadcasting** to multiple sockets, storing data **associated with each client**, and **asynchronous I/O**
- It has two parts:
  - A **client-side** library that runs in the browser
  - A **server-side** library for Node.js
- Can be installed with the **npm** tool

Adapted from http://socket.io/get-started/chat/

Building a chat room with Socket.IO in 20 minutes

Learning the basics of Socket.IO through a chat application!

Source code: https://github.com/mtyiu/socket-io-chat

Create an Express application skeleton

- Let's use the **Express** framework for simplicity
- Create an Express application called "`socket-io-chat`" and install dependencies:

```
$ express socket-io-chat
(Output omitted)
$ cd socket-io-chat
$ npm install
(Output omitted)
```

- Test if you can start the server properly:

```
$ npm start
```

Visit http://127.0.0.1:3000/

Lab 1: Display chat room UI

- **Goal:** Set up a route for displaying the UI
- Download the following files:
  - https://raw.githubusercontent.com/mtyiu/socket-io-chat/init/.gitignore
    - Put at the root directory
  - https://raw.githubusercontent.com/mtyiu/socket-io-chat/init/views/index.html
    - Put at the folder "`views/`"
- Let's serve the page "index.html" at the route "/"
  - Modify "`routes/index.js`"

Lab 1: Display chat room UI

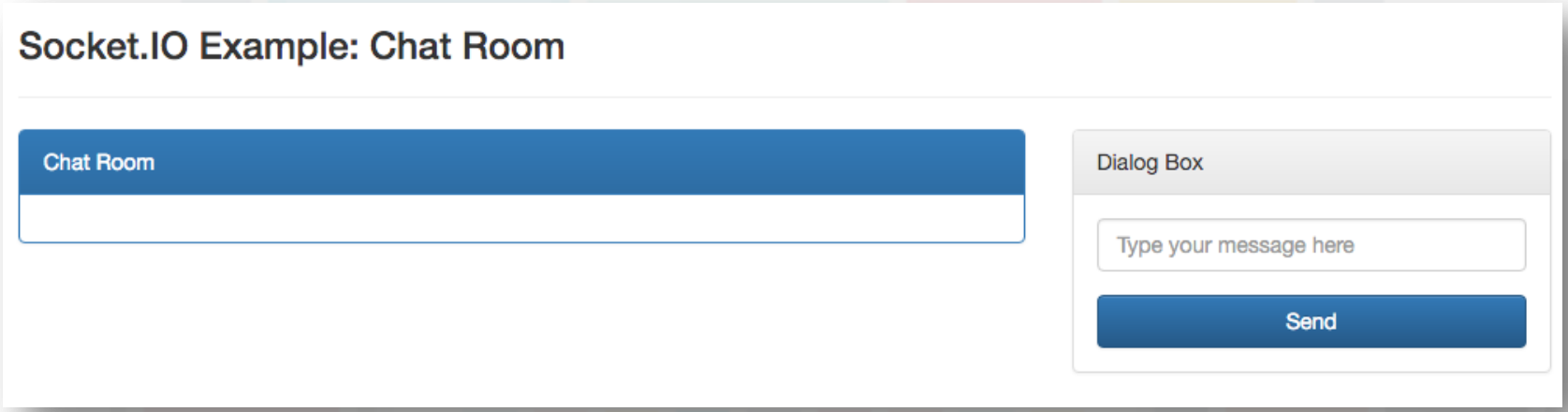
- Let's serve the page "index.html" at the route "/"
  - Modify "**routes/index.js**"

```
var express = require('express');
var path = require('path');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
    res.sendFile( path.resolve( __dirname +
'/../views/index.html' ) );
});

module.exports = router;
```

routes/index.js

Lab 1: Display chat room UI

- Restart the server and test if you can view the page at http://127.0.0.1:3000/



Lab 2: Integrate Socket.IO

- **Goal:** Integrate Socket.IO into our chat room
- Socket.IO is composed of two parts:
  - A server that integrates with (or mounts on) the **Node.JS HTTP Server**: **`socket.io`**
  - A client library that loads on the browser side: **`socket.io-client`**
    - This library is served to the client **automatically**
- Before using the library, we need to install it using npm

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

  - That will **install the module** and **add the dependency** to **`package.json`**

Lab 2: Integrate Socket.IO

- Integrate Socket.IO into **bin/www**

```
// ... (omitted)

var io = require( 'socket.io' )( server );
io.on( 'connection', function( socket ) {
    console.log( 'New user connected' );
} );
```

Add these lines to the end of the file.

bin/www

Lab 2: Integrate Socket.IO

- Integrate Socket.IO into **bin/www**

```
// ... (omitted)

var io = require( 'socket.io' )( server );
io.on( 'connection', function( socket ) {
    console.log( 'New user connected' );
} );

bi
```

Initialize a **socket.io** instance by passing the **server** object.

Listen on the **connection** event for incoming sockets.

The signature of the event listener is:

```
function (socket) { /* ... */ }
```

Lab 2: Integrate Socket.IO

- Integrate Socket.IO into **views/index.html**

```
<!-- ... (omitted) ... →
        <script src="/socket.io/socket.io.js"></script>
        <script>
            var socket = io();
        </script>
    </body>
</html>
```

Add these lines before **</body>**.

views/index.html

  - The first line loads the **socket.io-client** library which exposes an **io** global
  - Call **io()** without specifying any URL means to connect to the host that serves the page
- Now reload the server and refresh the web page

Lab 2: Integrate Socket.IO

- Try opening several tabs
- Can you see the message "**New user connected**" in the terminal?
- Each socket also fires a special **disconnect** event:

```
// ... (omitted)
var io = require( 'socket.io' )( server );
io.on( 'connection', function( socket ) {
    console.log( 'New user connected' );
    socket.on( 'disconnect', function() {
        console.log( 'User disconnected' );
    } );
} );
```

bin/www

Add these lines into **bin/www** and reload the server. You can see "**User disconnected**" upon each disconnection.

Lab 3: Single chat room *(client side)*

- You can send (or emit) and receive any events, with any data in Socket.IO
- Let's emit an "**chat**" event when the user types in a message
- Modify the last **<script>** tag in **views/index.html**:

```
<script>
    var socket = io();
    var $m = $( '#m' );
    var $messages = $( '#messages' );
    $( '#form' ).on( 'submit', function( e ) {
        e.preventDefault();
        socket.emit( 'chat', $m.val() );
        $m.val( '' );
    } );
</script>
```

views/index.html

Lab 3: Single chat room *(client side)*

- You can send (or emit) and receive any events, with any data in Socket.IO
- Let's emit an "**chat**" event when the user types in a message
- Modify the last **<script>** tag in **views/index.html**:

```
<script>
    var socket = io();
    var $m = $( '#m' );
    var $messages = $( '#messages' );
    $( '#form' ).on( 'submit', function( e ) {
        e.preventDefault();
        socket.emit( 'chat', $m.val() );
        $m.val( '' );
    } );
</script>
```

Get the DOM element using **querySelector()**.

Add an event listener for the form's **submit** event.

Emit a "**chat**" event with the message (**$m.val()**) as the data with Socket.IO

x.html

Lab 3: Single chat room *(server side)*

- Use `socket.on( <event>, function( data ) { /* ... */ } )` to handle our newly defined event

```
// ... (omitted)
var io = require( 'socket.io' )( server );
io.on( 'connection', function( socket ) {
    console.log( 'New user connected' );
    socket.on( 'disconnect', function() {
        console.log( 'User disconnected' );
    } );
    socket.on( 'chat', function( data ) {
        console.log( 'Message: ' + data );
        io.emit( 'chat', data );
    } );
} );
```

bin/www

Add these lines into **bin/www** and reload the server. You can see the message from the client upon each form submit.

Lab 3: Single chat room *(server side)*

- Use `socket.on( <event>, function( data ) { /* ... */ } )` to handle our newly defined event

```
// ... (omitted)
var io = require( 'socket.io' )( server );
io.on( 'connection', function( socket ) {
    console.log( 'New user connected' );
    socket.on( 'disconnect', function() {
        console.log( 'User disconnect
    } );
    socket.on( 'chat', function( data ) {
        console.log( 'Message: ' + data );
        io.emit( 'chat', data );
    } );
} );
```

bin/www

A **chat** event is emitted to all connected clients with the **data**.

Lab 3: Single chat room *(client side)*

- Listen to the **chat** event in the client side:

```
<script>
    var socket = io();
    var $m = $( '#m' );
    var $messages = $( '#messages' );
    $( '#form' ).on( 'submit', function( e ) {
        e.preventDefault();
        socket.emit( 'chat', $m.val() );
        $m.val( '' );
    } );
    socket.on( 'chat', function( data ) {
        $messages.append( '<li>' + data + '</li>' );
    } );
</script>
```

views/index.html

Lab 3: Single chat room *(client side)*

- Listen to the **chat** event in the client side:

```
<script>
    var socket = io();
    var $m = $( '#m' );
    var $messages = $( '#messages' );
    $( '#form' ).on( 'submit',
        tDefault();
        mit( 'chat', $
        '' );
    } );
    socket.on( 'chat', function( data ) {
        $messages.append( '<li>' + data + '</li>' );
    } );
</script>
```

views/index.html

Set up a **chat** event listener to the **socket**.

The signature of the event listener is:

```
function ( data ) { /* ... */ }
```

Display the incoming message by updating the DOM tree.

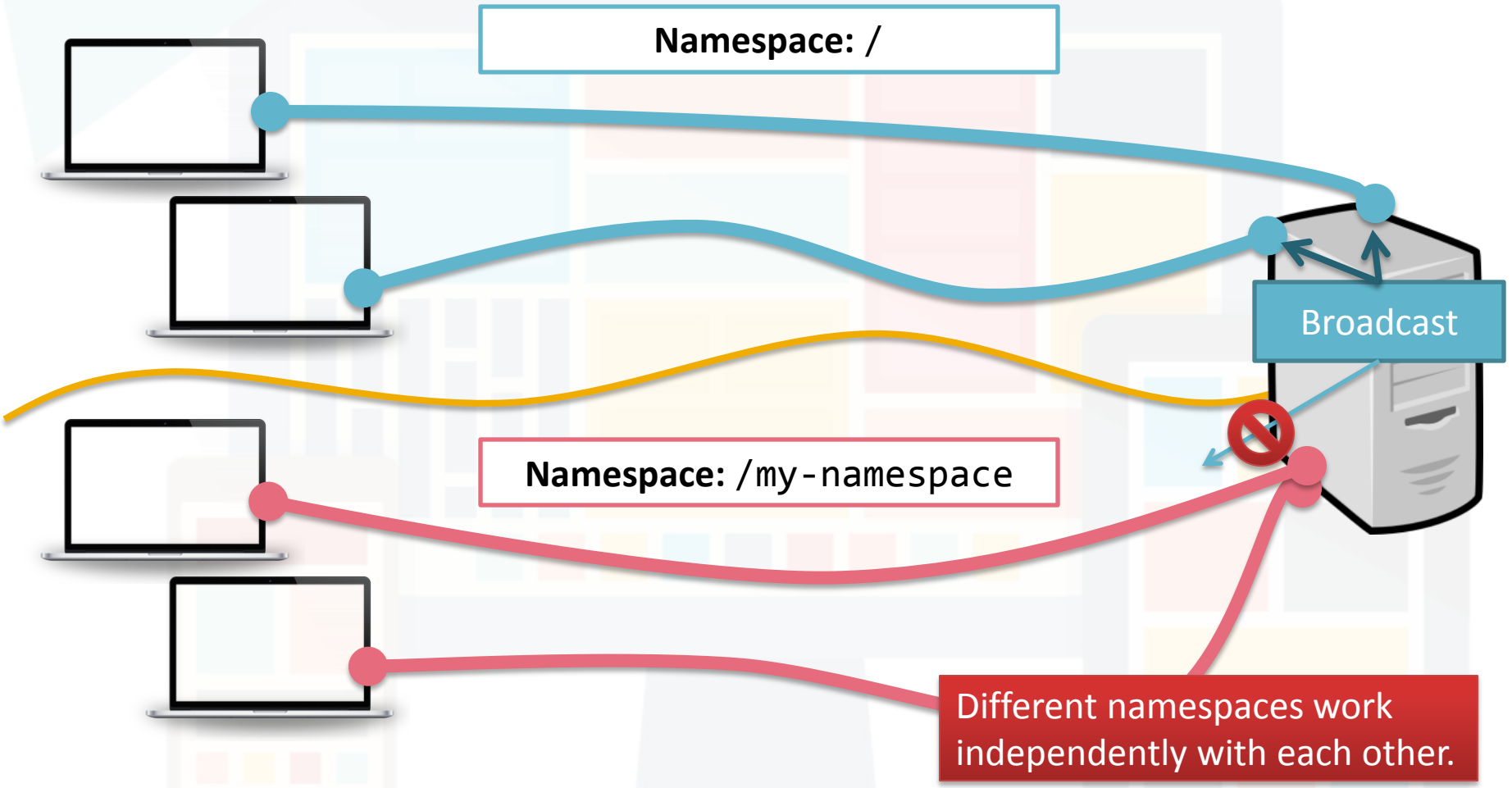
Namespaces and rooms in Socket.IO

We can broadcast among clients in the same namespace / room only!

Namespaces

- Socket.IO allows you to "**namespace**" your sockets, which essentially means assigning different **endpoints** or **paths**
- Useful for
  - **Minimizing** the number of **resources** (e.g., TCP connections)
  - Introducing **separation** between **communication channels**
- The default namespace is "**/**"
  - The clients connect to this namespace by default
  - The server listens to this namespace by default

Namespaces


Namespace: /
Broadcast
Namespace: /my-namespace
Different namespaces work independently with each other.

Custom namespaces

- To set up a custom namespace, call the **of** function on the **server-side**:

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

- On the **client side**, specify the namespace in the **io** function:

```
var socket = io( '/my-namespace' );
```

For your information, my implementation does not use custom namespaces to separate different sessions. I use "room" instead!

Rooms

- Within each namespace, you can also define arbitrary channels (denoted as "**room**") that sockets can **`join`** and **`leave`**
- To assign the sockets into different rooms on the server side:

```
io.on( 'connection', function( socket ) {
    socket.join( 'some room' );
} );
```

Of course, you can also call `join()` (i.e., subscribe the socket to a given channel) when other events are emitted, e.g., "**`register`**" event

- To broadcast or emit, call **`to()`** or **`in()`**:

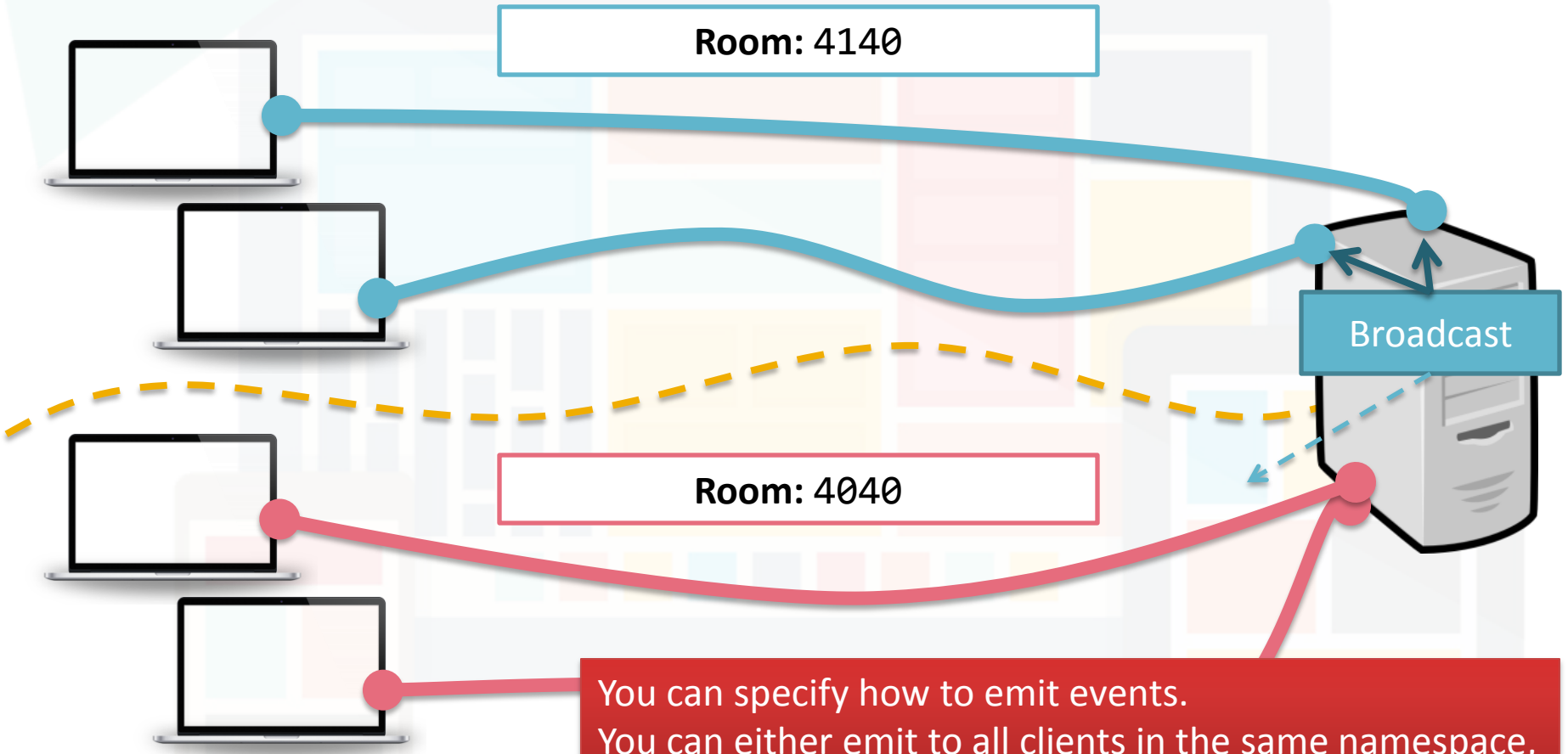
```
io.to( 'some room' ).emit( 'some event' );
```

- To leave a channel:

```
socket.leave( 'some room' );
```

  - This is automatically done upon disconnection

Rooms under the same namespace



Lab 4: Multiple chat rooms

- Multiple chat rooms is extremely easy in Socket.IO!
- Let's assign each chat room with an ID
  - Modify the routes as follows:

Route	Action
/	Redirect to a new room
/[room ID]	Display the chat room with the corresponding ID

Lab 4: Multiple chat rooms *(server side)*

```
var express = require('express');
var path = require('path');
var router = express.Router();

router.get( '/', function( req, res ) {
    var id = Math.floor( Math.random() * 10000 );
    res.redirect( '/' + id );
} );

router.get( '/:id([0-9]+)', function( req, res ) {
    res.sendFile( path.resolve( __dirname +
'/../views/index.html' ) );
} );

module.exports = router;
```

routes/index.js

Lab 4: Multiple chat rooms *(server side)*

```
var express = require('express');
var path = require('path');
var router = express.Router();

router.get( '/', function( req, res ) {
    var id = Math.floor( Math.random() * 10000 );
    res.redirect( '/' + id );
} );

router.get( '/:id([0-9]+)', function( req, res ) {
    res.sendFile( path.resolve( __dirname +
'/../views/index.html' ) );
} );

module.exports = router;
```

Generate a random room ID and redirect to the corresponding route.

Set up a route with route parameter "**id**", which displays the chat room UI.

routes/index.js

Lab 4: Multiple chat rooms

- Do you remember that you need to assign a socket to a room with `socket.join()`?
  - How can the server know which room a socket should join?
  - A simple approach is to use a "register" event to determine which room to join
- Client first retrieves its room ID from the URL, then emit a register event on the socket

Lab 4: Multiple chat rooms *(client side)*

```
<script>
    if ( ! Array.prototype.last ) {
        Array.prototype.last = (function() {
            return this[ this.length - 1 ];
        });
    }
    var id = window.location.href.split( '/' ).last();

    var socket = io();
    socket.emit( 'register', id );

    // omitted...
</script>
```

views/index.html

Retrieve room ID by parsing the current URL.

Before using the socket, emit a register event with the room ID.

Lab 4: Multiple chat rooms *(server side)*

```
var io = require( 'socket.io' )( server );
io.on( 'connection', function( socket ) {
    var id;
    console.log( 'New user connected' );

    socket.on( 'disconnect', function() {
        console.log( 'User disconnected' );
    } );

    socket.on( 'register', function( data ) {
        console.log( 'User registered. Room ID = ' + data );
        id = data;
        socket.join( data );
    } );

    socket.on( 'chat', function( data ) {
        console.log( 'Message: ' + data );
        io.to( id ).emit( 'chat', data );
    } );
} );
```

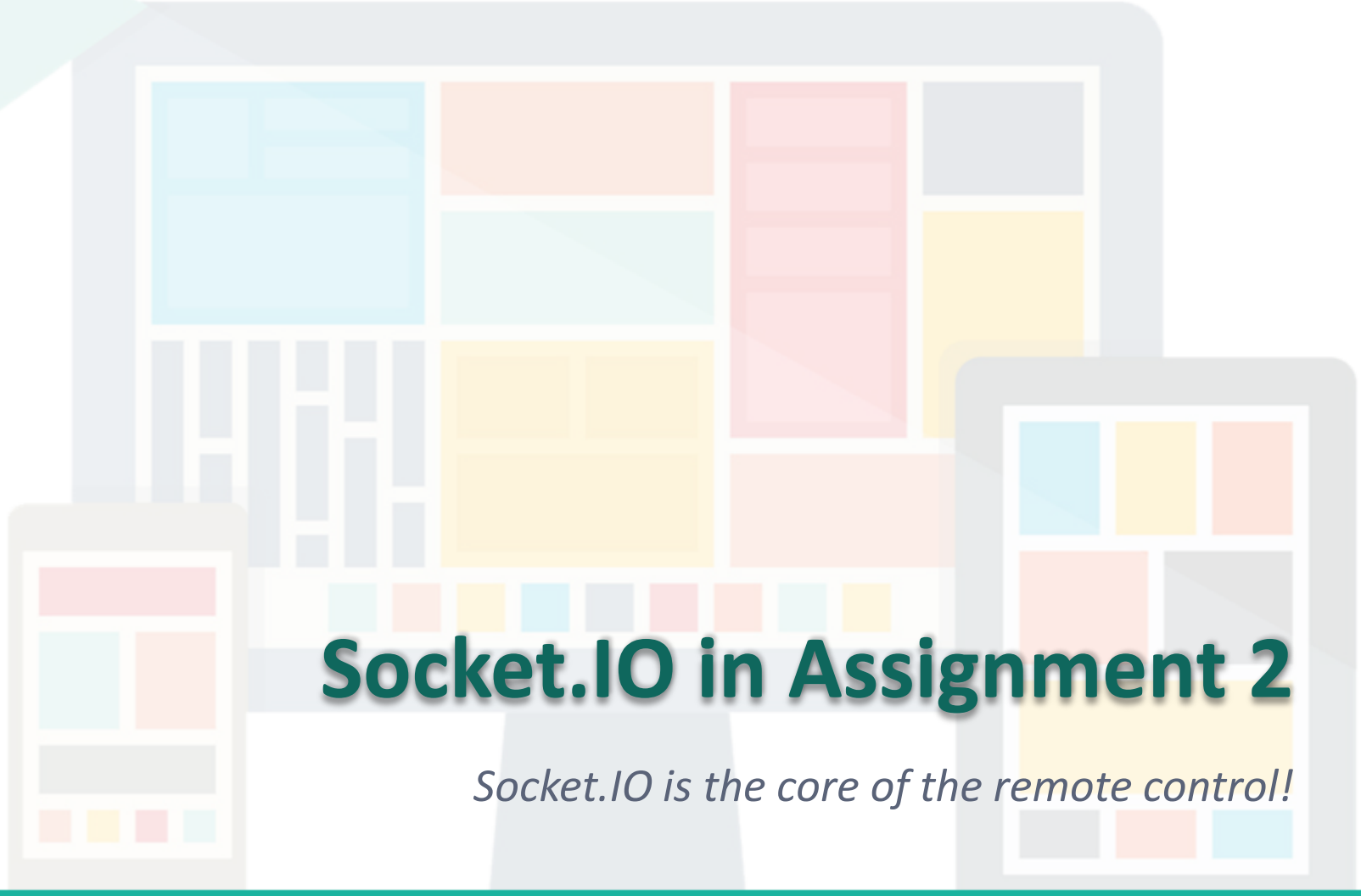
Store the room ID when it receives a register event. Remember to call **`socket.join()`** after the client registered.

Instead of broadcasting the message to all connected sockets, we only broadcast to sockets in the same room.

bin/www

Lab 4: Multiple chat rooms

- Now you can restart the server and test if you can have multiple chat rooms working independently
  - The implementation of multiple sessions in Assignment 2 is very similar!
  - It only takes a few lines of code! Simple enough?
- Lab 5 is to deploy your chat room on Heroku
  - Refer to the tutorial notes on Heroku



Socket.IO in Assignment 2

Socket.IO is the core of the remote control!

Socket.IO in Assignment 2

- Socket.IO is used for
  - **Connecting** the clients to the server
  - **Broadcasting control signals** to the desktop clients
  - **Synchronizing** the playlist
- Emitted events in my implementation *(for your reference only)*
  - `register` (***data***: session ID) – Assign a socket to a room
  - `download` (***data***: `null` or `playlist`) – Download a playlist from the server
  - `command` (***data***: control signal to the player)
  - `add` / `remove` (***data***: video ID to be added or removed)
  - **Feel free to design your own protocol!**

References

- Get Started: Chat application
  - http://socket.io/get-started/chat/
- Server API:
  - http://socket.io/docs/server-api/
- Client API:
  - http://socket.io/docs/client-api/
- Rooms and Namespaces:
  - http://socket.io/docs/rooms-and-namespaces/

– End –