Storing Data with Mongo DB

Prof. Fabio Ciravegna
Department of Computer Science
The University of Sheffield
f.ciravegna@shef.ac.uk

MongoDB

- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling

https://docs.mongodb.com/getting-started/shell/introduction/

Documents

- A record in MongoDB is a document, which is a data structure composed of field and value pairs.
- MongoDB documents are similar to JSON objects.
- The values of fields may include other documents, arrays, and arrays of documents

SQL Records -> Mongos' Documents

Collections

- MongoDB stores documents in collections. Collections are analogous to tables in relational databases.
  - Unlike a table, however, a collection does not require its documents to have the same schema
- In MongoDB, documents stored in a collection must have a **unique _id** field that acts as a **primary key**

SQL Relations -> Mongos' Collections

The University Of Sheffield.

A restaurant record example


```
{
   "_id" : ObjectId("54c955492b7c8eb21818bd09"),
   "address" : {
      "street" : "2 Avenue",
      "zipcode" : "10075",
      "building" : "1480",
      "coord" : [ -73.9557413, 40.7720266 ]
   },
   "borough" : "Manhattan",
   "cuisine" : "Italian",
   "grades" : [
      {
         "date" : ISODate("2014-10-01T00:00:00Z"),
         "grade" : "A",
         "score" : 11
      },
      {
         "date" : ISODate("2014-01-16T00:00:00Z"),
         "grade" : "B",
         "score" : 17
      }
   ],
   "name" : "Vella",
   "restaurant_id" : "41704620"
}
```

BSON

- Mongo's documents are internally represented as binary JSON
  - this is why in express you are likely to have to include the nam module on bson

https://www.slideshare.net/mdirolf/introduction-to-mongodb

Advantages

Faster process

Open Source

Sharding

mongoDB

Schemaless

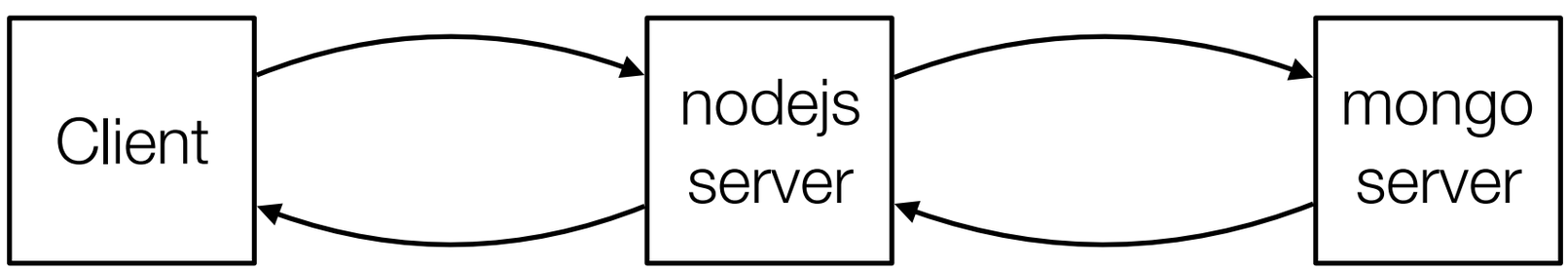
Document based

No SQL Injection

https://www.studytonight.com/mongodb/advantages-of-mongodb

A separate process

- Mongo, as any db system, must run in a separate process from your main nodejs server
  - remember: nodes is fast and scalable but no long running process is to be run on its single thread
  - create a separate process for that and connect using the appropriate library
    - Mongoose in our case



The University Of Sheffield.

mongoose public

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment.

slack 10/386 | build passing | npm package 5.0.9

```
npm install mongoose
```

```
// Using Node.js `require()`
const mongoose = require('mongoose');
```

Creating a Database

- To create a database in MongoDB,
  - start by creating a MongoClient object,
  - then specify a connection URL with
    - the correct ip address and
    - the name of the database you want to create.
- MongoDB will create the database if it does not exist, and make a connection to it.

https://www.w3schools.com/nodejs/nodejs_mongodb_create_db.asp

The University Of Sheffield.

Example

Create a database called "mydb":

typically running on 27017 or 27019

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

The University Of Sheffield.

Schemas

- Although mongo is schemaless, it helps to define schemas in order to
  - validate data integrity of documents
  - legibility/maintenance
    - pretty much like in Javascript where although the same variable can contain different data structures, you end up specialising the variables to only one type
- Mongoose imposes a schema for document models

- A schema is the equivalent of a java interface
  - it must be implemented in a model before being used
- Unlike java interfaces however, it defines types and restrictions
- It is not an instance that can be used

```
var Character = new Schema(
    {
        first_name: {type: String},
        family_name: {type: String},
        dob: {type: Number},
        whatever: {type: String} //any other field
    }
);
```

The University Of Sheffield.

Aside from defining the structure of your documents and the types of data you're storing, a Schema handles the definition of:

- **Validators** (async and sync)
- **Defaults**
- **Getters**
- **Setters**
- **Indexes**
- **Middleware**
- **Methods** definition
- **Statics** definition
- **Plugins**
- **pseudo-JOINs**

Validation

Mongoose provides built-in and custom validators, and synchronous and asynchronous validators. It allows you to specify both the acceptable range or values and the error message for validation failure in all cases.

The built-in validators include:

- All SchemaTypes have the built-in required validator. This is used to specify whether the field must be supplied in order to save a document.
- Numbers have min and max validators.
- Strings have:
  - enum: specifies the set of allowed values for the field.
  - match: specifies a regular expression that the string must match.
  - maxlength and minlength for the string.

Validation Example

max string length

```
var Character = new Schema(
    {
        first_name: {type: String, required: true, max: 100},
        family_name: {type: String, required: true, max: 100},
        dob: {type: Number, required: true, max: 2020},
        whatever: {type: String} //any other field
    }
);
```

max value

see details at http://mongoosejs.com/docs/validation.html

The University Of Sheffield.

Validation example

```
var breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12
    required: [true, 'Why no eggs?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea', 'Water',]
  }
});
```

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

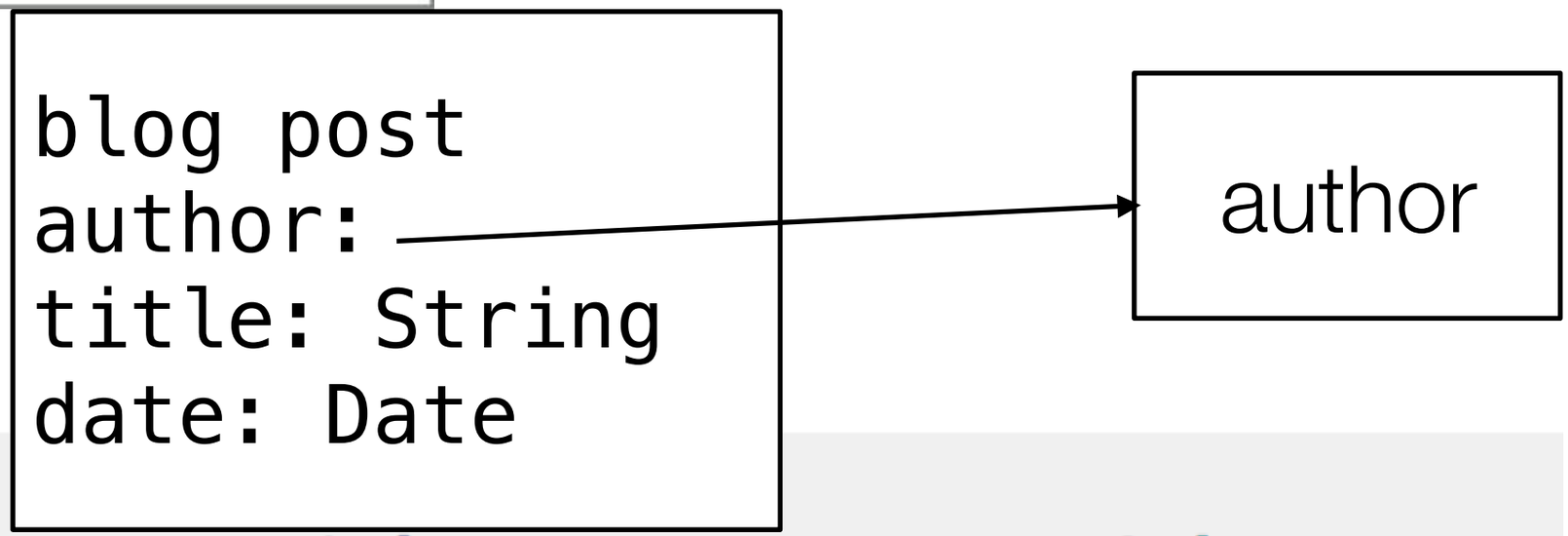
Virtual properties

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

- Document properties that you can get and set but that do not get persisted to MongoDB
- The getters are useful for formatting or combining fields,
  - It is easier and cleaner and uses less disk space
  - it allows for dynamic properties
- Examples:
  - a full name virtual property starting from concrete fields called first name and last name
  - (Dynamic): the age of a person computed from the current year and date of birth

```
// Virtual for age of a person
Character.virtual('age')
.get(function () {
  const currentDate = new Date().getTime();
  return currentDate-this.date_of_birth;
});
```

Objects in fields




```
const Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;

const BlogPost = new Schema({
 author: ObjectId,
 title: String,
 body: String,
 date: Date
});
```

- `ObjectId`: Represents specific instances of a model in the database. For example, a book might use this to represent its author object. This will actually contain the unique ID (`_id`) for the specified object.

Middleware : pre/post operations

```
const Comment = new Schema({
  name: { type: String, default: 'hahaha' },
  age: { type: Number, min: 18, index: true },
  bio: { type: String, match: /[a-z]/ },
  date: { type: Date, default: Date.now },
  buff: Buffer
});

// a setter
Comment.path('name').set(function (v) {
  return capitalize(v);
});

// middleware
Comment.pre('save', function (next) {
  notify(this.get('email'));
  next();
});
```

Used to perform operations before saving (for example to notify an administrator)

Models implement Schemas

- The Schema allows you to define the fields stored in each document along with their validation requirements and default values.
- Schemas are then "compiled" into models using the **`mongoose.model()`** method.

- Once you have a model you can use it to
  - find,
  - create,
  - update,
  - delete

} instances of that model (i.e. records in the db)

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

Instances

new <ModelName>({<fields>}) creates an instance of a model

```
var character = new Character({
    first_name: 'Mickey',
    family_name: 'Mouse',
    dob: 1908
});
```

then save it into the database (with callback)

```
character.save(function (err, results) {
    console.log(results._id);
});
```

Or you can do it in steps

Once we have our model, we can then instantiate it, and save it:

```
const instance = new MyModel();
instance .field= 'hello';
instance.save(function (err) {
  //  check if there are errors
});
```

Fill the instance by assigning field by field

Searching

Searching for records

You can search for records using query methods, specifying the query conditions as a JSON document. The code fragment below shows how you might find all athletes in a database that play tennis, returning just the fields for athlete *name* and *age*. Here we just specify one matching field (sport) but you can add more criteria, specify regular expression criteria, or remove the conditions altogether to return all athletes.

```
var Athlete = mongoose.model('Athlete', yourSchema);

// find all athletes who play tennis, selecting the 'name' and 'age' fields
Athlete.find({ 'sport': 'Tennis' }, 'name age', function (err, athletes) {
  if (err) return handleError(err);
  // 'athletes' contains the list of athletes that match the criteria.
})
```

If you specify a callback, as shown above, the query will execute immediately. The callback will be invoked when the search completes.

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

The University Of Sheffield.

Example

```
var character = new Character({
    first_name: 'Mickey',
    family_name: 'Mouse',
    dob: 1908
});
console.log('dob: ‘+character.dob);

character.save(function (err, results) {
    console.log(results._id);
});
```

```
Character.find({first_name: some strings, family_name: some strings},
    'first_name family_name dob age’,  // these are the fields to return
    function (err, characters) {
        if (err)
            res.status(500).send('Invalid data!’);
...
```

Callback Parameters

If you specify a callback, as shown above, the query will execute immediately. The callback will be invoked when the search completes.

> **Note:** All callbacks in Mongoose use the pattern `callback(error, result)`. If an error occurs executing the query, the `error` parameter will contain an error document, and `result` will be null. If the query is successful, the `error` parameter will be null, and the `result` will be populated with the results of the query.

If you don't specify a callback then the API will return a variable of type Query. You can use this query object to build up your query and then execute it (with a callback) later using the `exec()` method.

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

Querying step by step

```
// find all athletes that play tennis
var query = Athlete.find({ 'sport': 'Tennis' });

// selecting the 'name' and 'age' fields
query.select('name age');

// limit our results to 5 items
query.limit(5);

// sort by age
query.sort({ age: -1 });

// execute the query at a later time
query.exec(function (err, athletes) {
  if (err) return handleError(err);
  // athletes contains an ordered list of 5 athletes who play Tennis
})
```

Above we've defined the query conditions in the `find()` method. We can also do this using a `where()` function, and we can chain all the parts of our query together using the dot operator (.) rather than adding them separately. The code fragment below is the same as our query above, with an additional condition for the age.

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

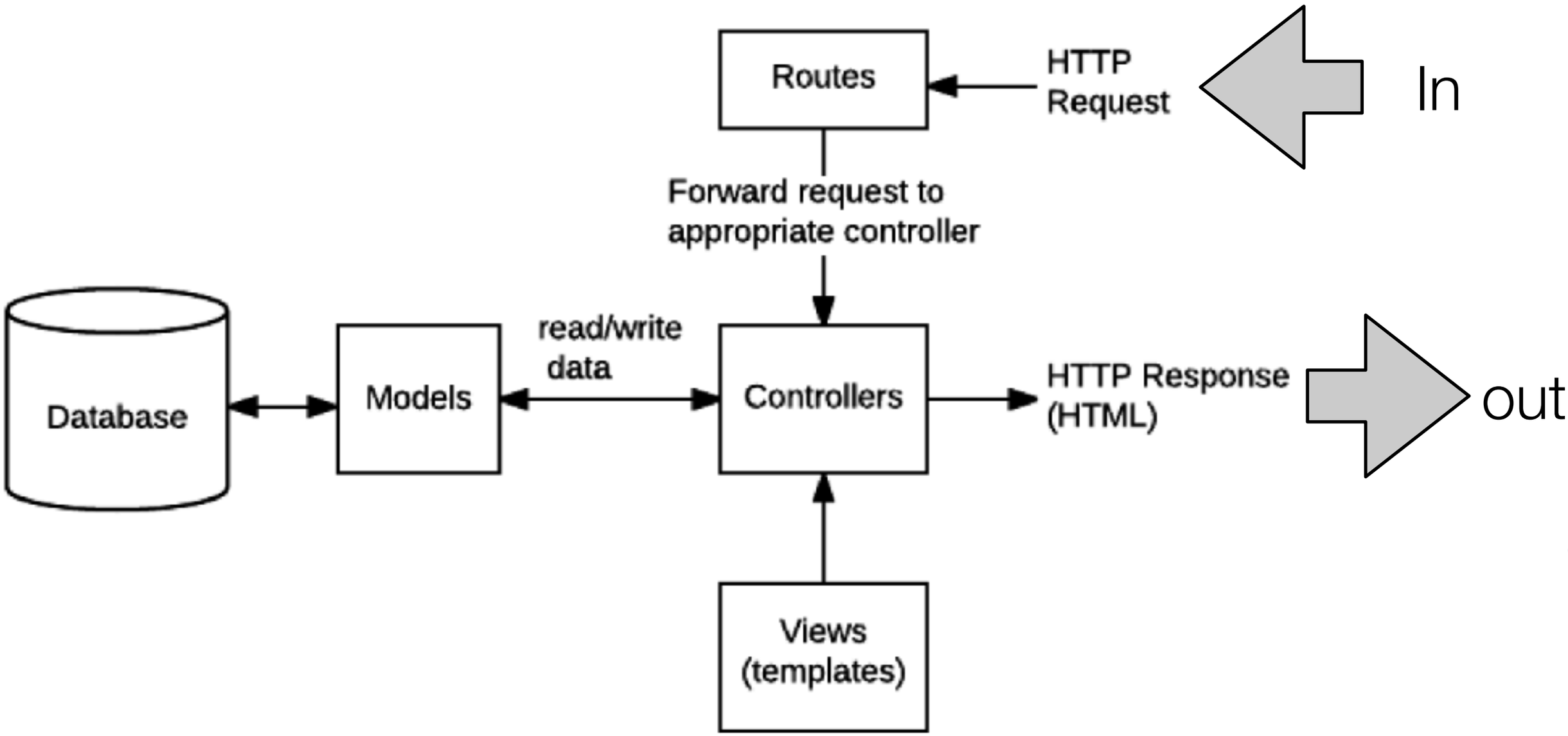
composing with dot

Above we've defined the query conditions in the `find()` method. We can also do this using a `where()` function, and we can chain all the parts of our query together using the dot operator (.) rather than adding them separately. The code fragment below is the same as our query above, with an additional condition for the age.

```
Athlete.
  find().
  where('sport').equals('Tennis').
  where('age').gt(17).lt(50).  //Additional where query
  limit(5).
  sort({ age: -1 }).
  select('name age').
  exec(callback); // where callback is the name of our callback function.
```

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

Typical project organisation



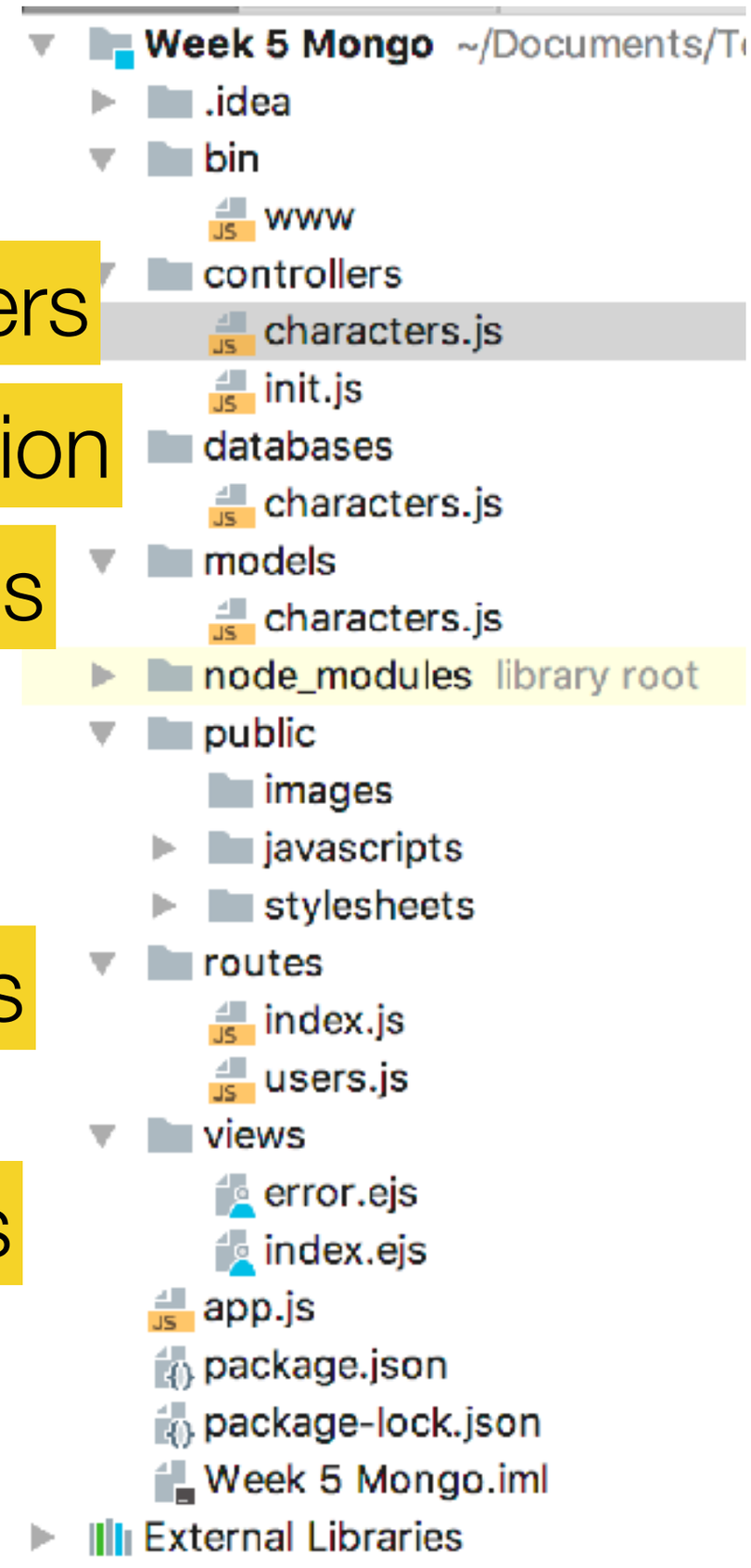
https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes

…Organisation

- Routes to forward the supported requests (and any information encoded in request URLs) to the appropriate controller functions.
- Controller functions to get the requested data from the models, create an HTML page displaying the data, and return it to the user to view in the browser.
- Views (templates) used by the controllers to render the data.
- Models: the declaration of the MongoDb equivalent of relations

In IntelliJ

- The program is organised in this way
- There is a database called ‘characters’
  - which has a model called ‘Character’ representing name, surname and year of birth of each character

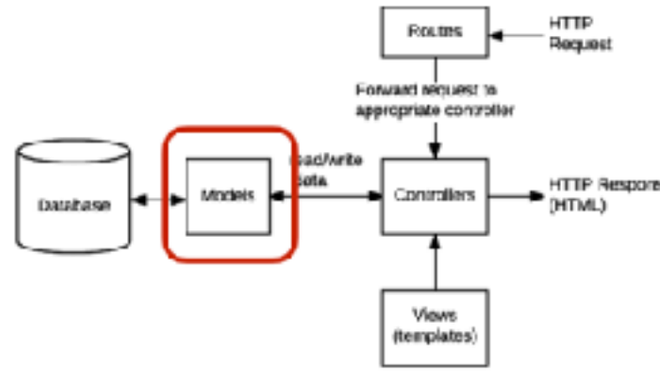


The University Of Sheffield.

routes/my_routes.js

```
// POST request to update Author.
router.post('/author/:id/update', author_controller.author_update_post);

// GET request for one Author.
router.get('/author/:id', author_controller.author_detail);

// GET request for list of all Authors.
router.get('/authors', author_controller.author_list);

/// GENRE ROUTES ///

// GET request for creating a Genre. NOTE This must come before route that d
router.get('/genre/create', genre_controller.genre_create_get);
```

Organising Models



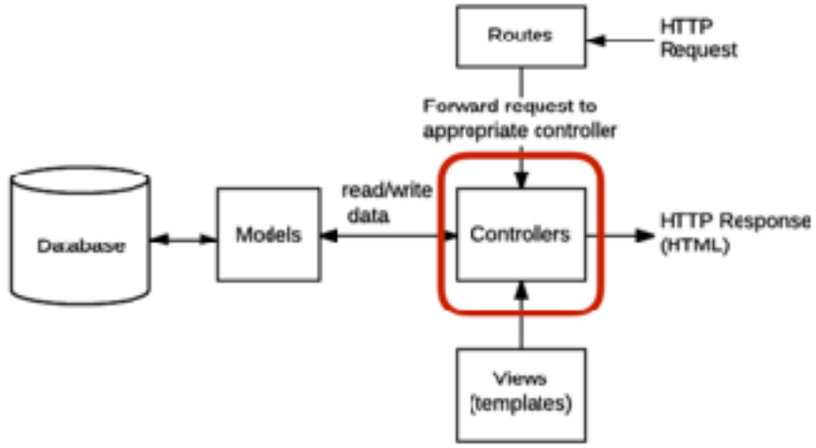
- It is recommended that you define just one model per file
  - and then export the model

```
/Project root
  /models
    author.js
    book.js
    bookinstance.js
    genre.js
```

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var Character = new Schema({
        first_name: {type: String, required: true, max: 100},
        family_name: {type: String, required: true, max: 100},
        dob: {type: Number},
        whatever: {type: String} });

var characterModel = mongoose.model('Character', Character );

module.exports = characterModel;
```

we will see how to import the model in a controller in a few slides

Controllers



- Controllers are the route-handler callback functions
  - it is suggest not to insert too much code into the routes file(s)
    - so to keep the code clean and separated
  - So `routes/index.js` could contain something like:

```
var author_controller = require('../controllers/authorController');

  ...

// GET request for list of all Authors.
router.get('/authors', author_controller.author_list);
```

Function defined in authorController

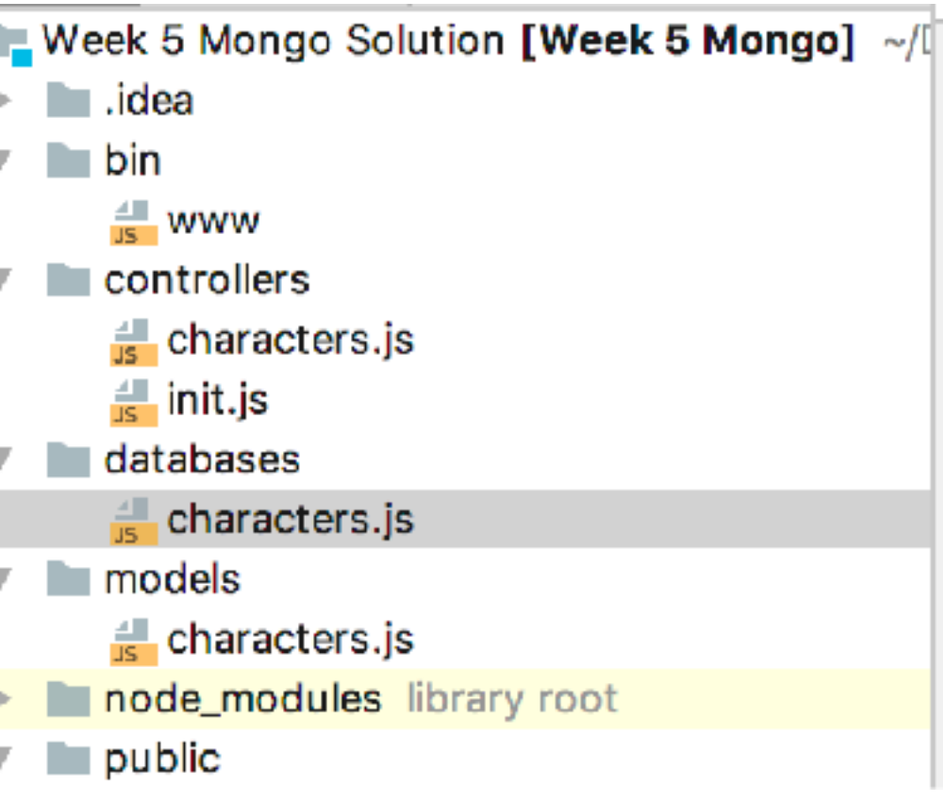
Controllers

- They Import the models and use the model as an object, e.g.:
  - then they define a list of exported functions to be used in the routes

```
var Author = require('../models/author');

// Display list of all Authors.
exports.author_list = function(req, res) {
    res.send('NOT IMPLEMENTED: Author list');
};

// Display detail page for a specific Author.
exports.author_detail = function(req, res) {
    res.send('NOT IMPLEMENTED: Author detail: ' + req.params.id);
```

In the lab

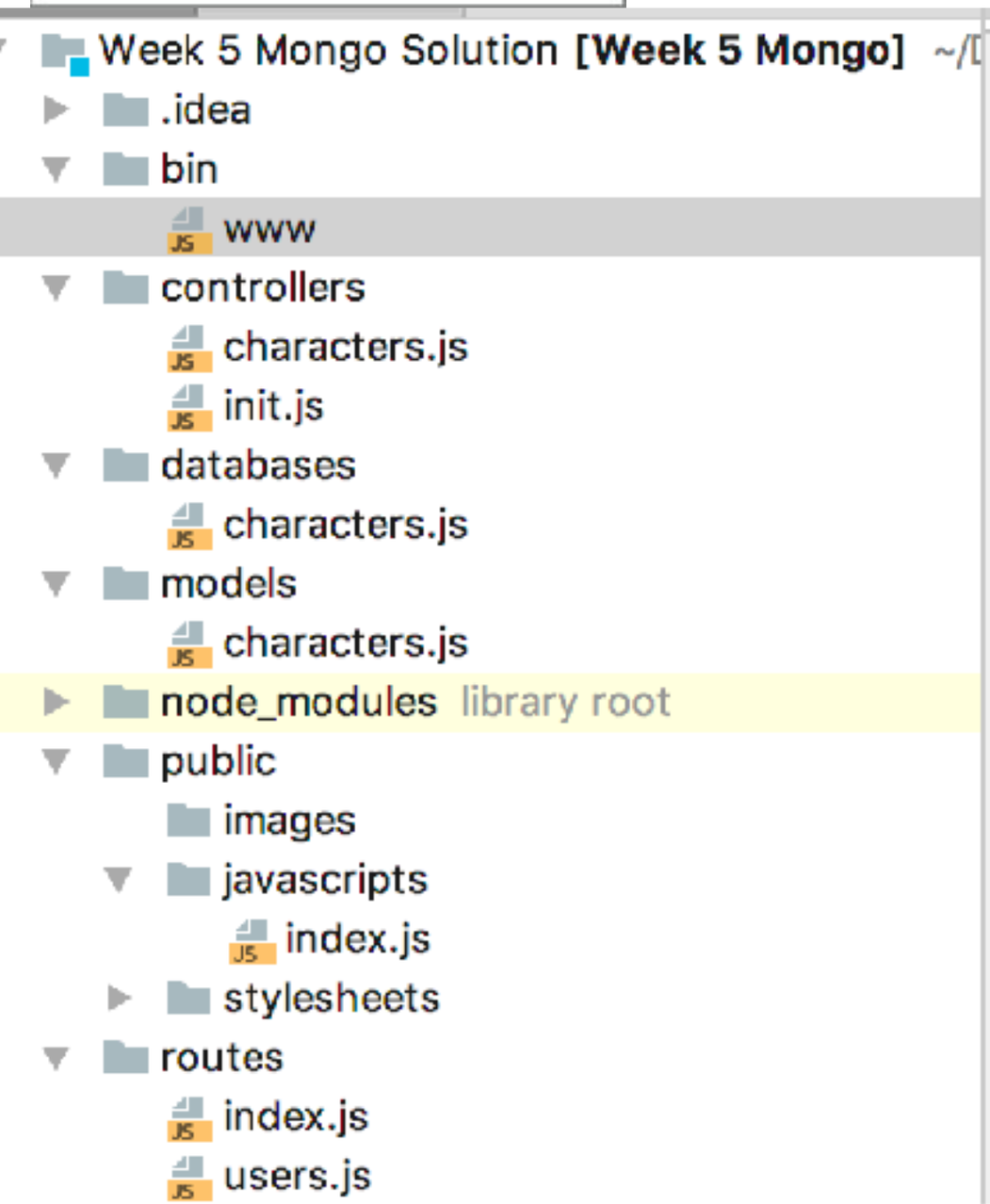
- create a js file called database.js under databases




```
var mongoose = require('mongoose');
var ObjectId = require('mongodb').ObjectID;
var bcrypt = require('bcryptjs');

//The URL which will be queried. Run "mongod.exe" for this
//var url = 'mongodb://localhost:27017/test';
var mongoDB = 'mongodb://localhost:27019/characters';

mongoose.Promise = global.Promise;
mongoose.connect(mongoDB);
var db = mongoose.connection;
//Bind connection to error event (to get notification of
db.on('error', console.error.bind(console, 'MongoDB conne
```

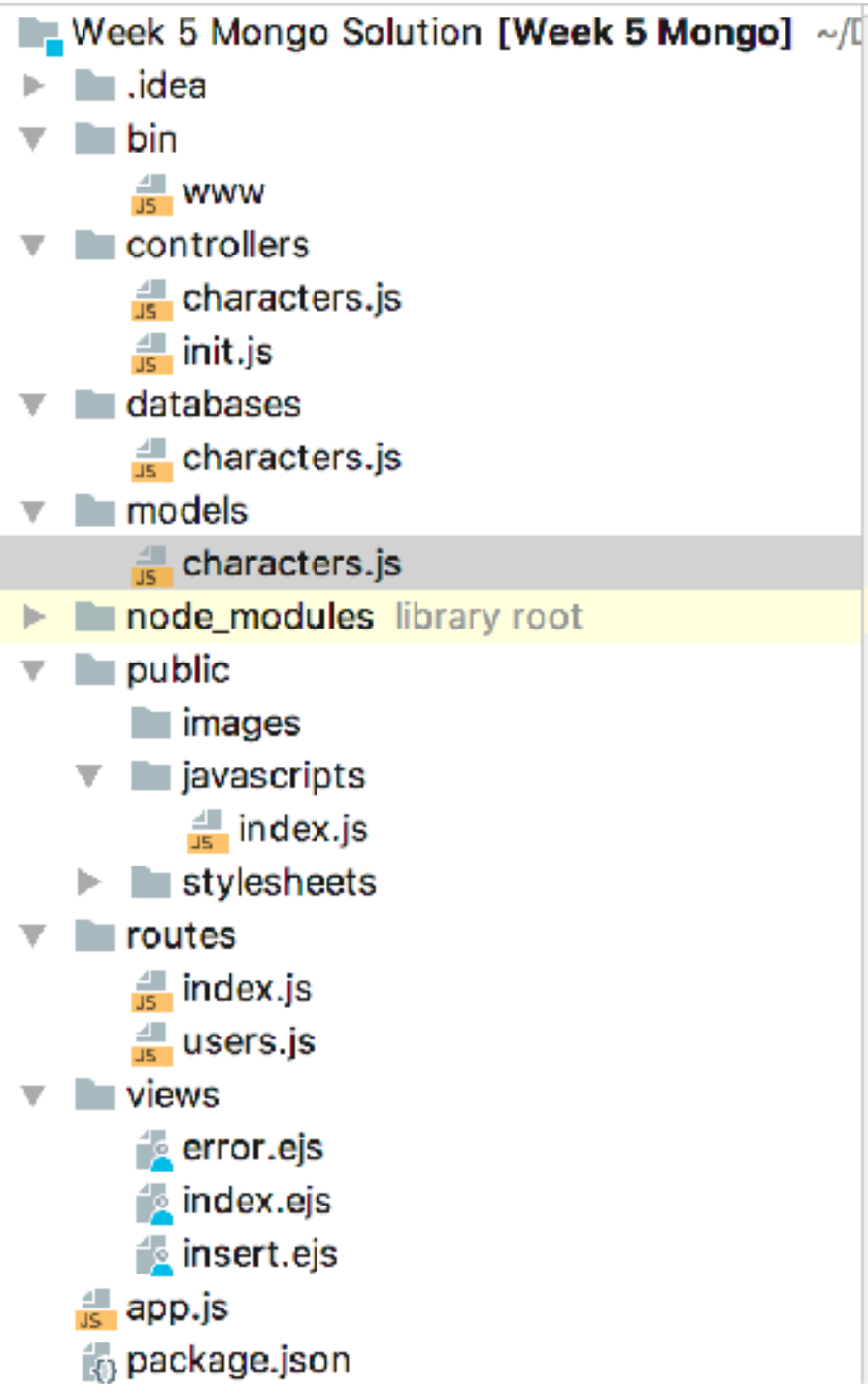
this will connect to the Mongo server and will create the characters database (if not existent)

Require the db in bin/www



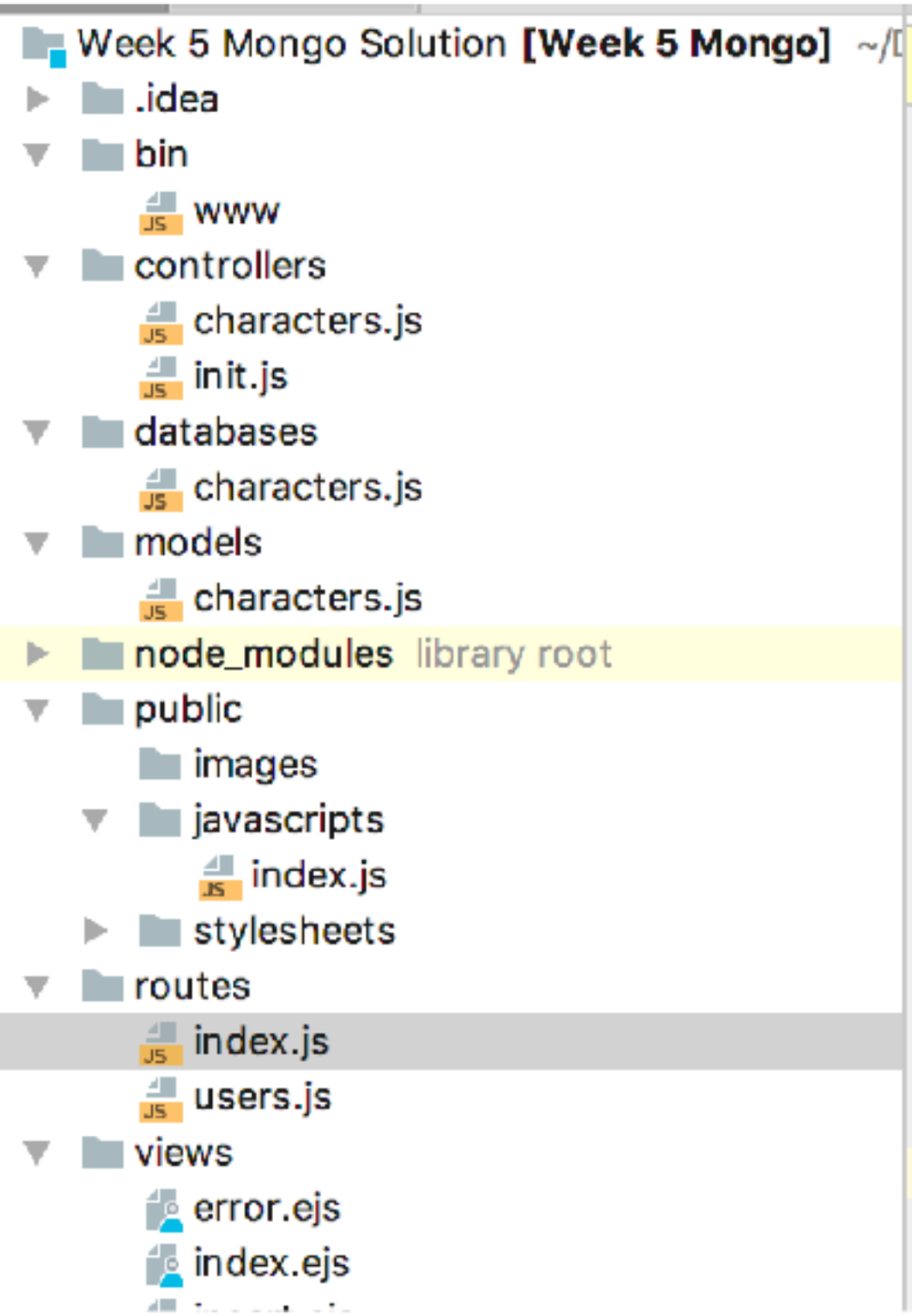

```
1   #!/usr/bin/env node
2
3   /**
4    * Module dependencies.
5    */
6
7   var app = require('../app');
8   var debug = require('debug')('week-5-mongo:server');
9   var http = require('http');
10  var database= require('../databases/characters')
11  /**
12   * Get port from environment and store in Express.
13   */
14
15  var port = normalizePort(process.env.PORT || '3003');
16  app.set('port', port);
17
18  /**
19   * Create HTTP server.
20   */
21
22  var server = http.createServer(app);
```

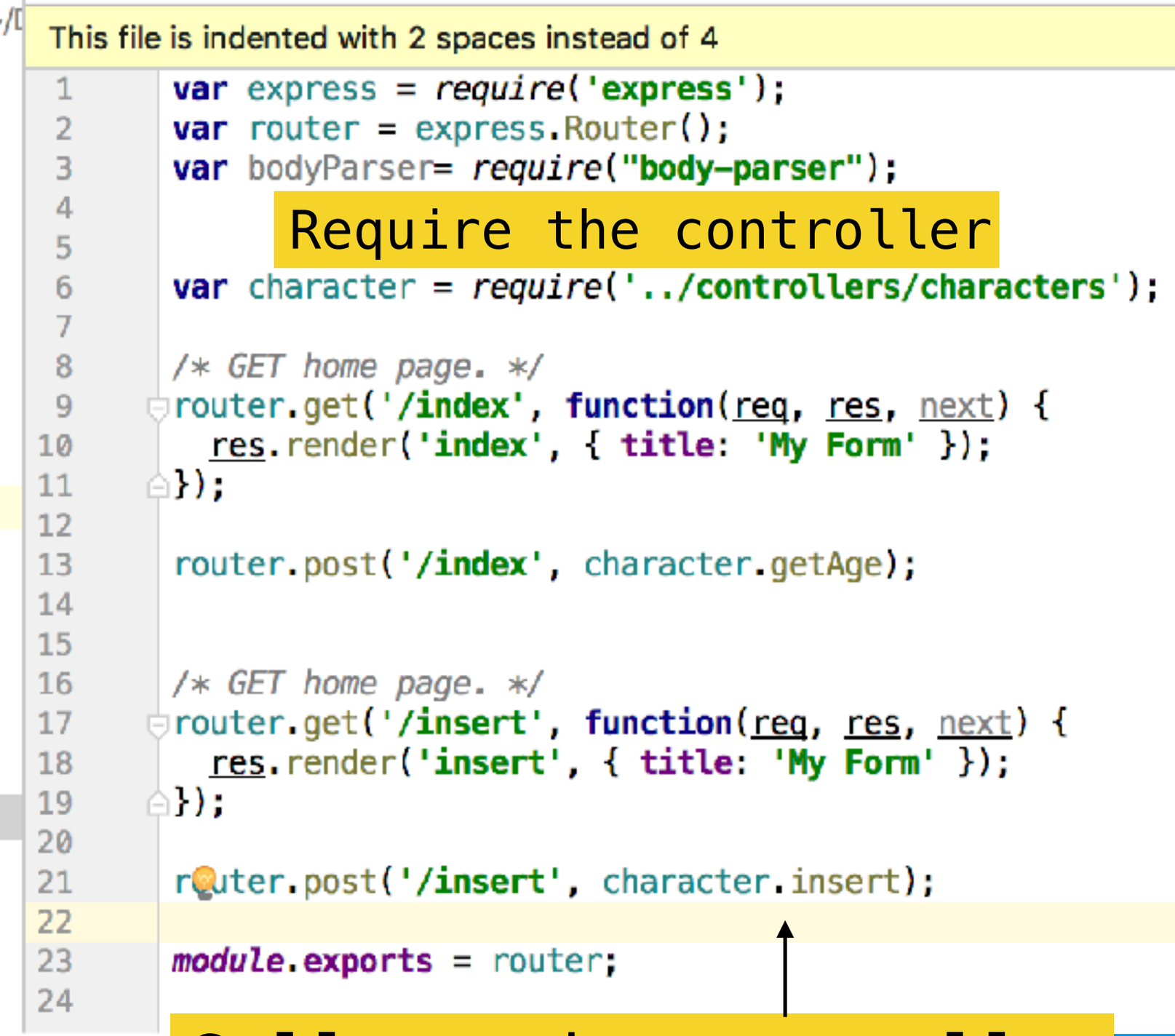
define a model for the data



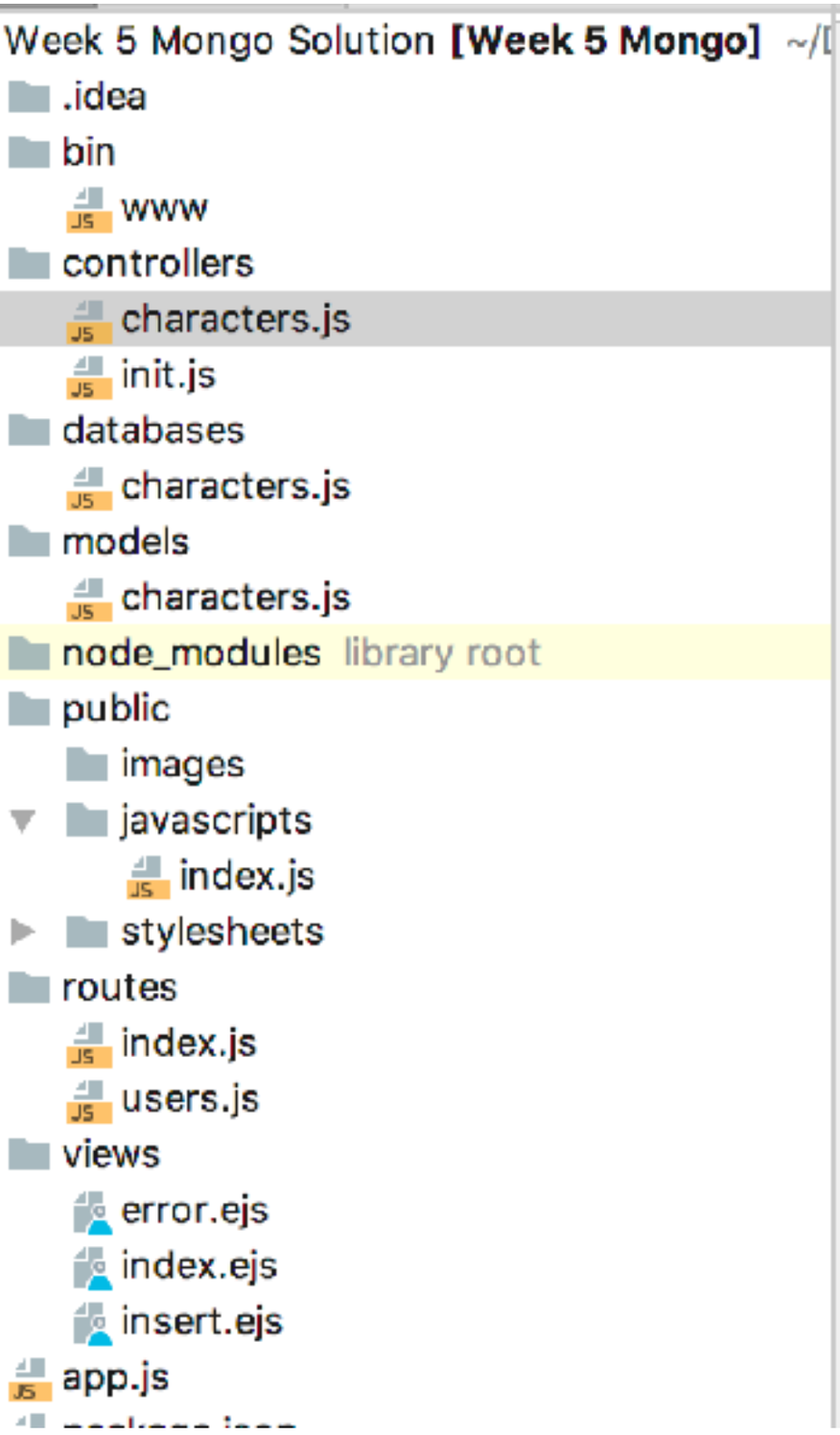

```
1  var mongoose = require('mongoose');
2
3  var Schema = mongoose.Schema;
4
5  var Character = new Schema(
6      {
7          first_name: {type: String, required: true, max: 100},
8          family_name: {type: String, required: true, max: 100},
9          dob: {type: Number},
10         whatever: {type: String} //any other field
11     }
12 );
13
14 // Virtual for a character's age
15 Character.virtual('age')
16     .get(function () {
17         const currentDate = new Date().getFullYear();
18         const result= currentDate - this.dob;
19         return result;
20     });
21
22 Character.set('toObject', {getters: true, virtuals: true});
23
24
25 var characterModel = mongoose.model('Character', Character );
26
27 module.exports = characterModel;
```

Define the Routes





The controller




```
1   var Character = require('../models/characters');
2
3
4   exports.insert = function (req, res) {
5       var userData = req.body;
6       if (userData == null) {
7           res.status(403).send('No data sent!')
8       }
9       try {
10          var character = new Character({
11              first_name: userData.firstname,
12              family_name: userData.lastname,
13              dob: userData.year
14          });
15          console.log('received: ' + character);
16
17          character.save(function (err, results) {
18              console.log(results._id);
19              if (err)
20                  res.status(500).send('Invalid data!');
21
22              res.setHeader('Content-Type', 'application/json');
23              res.send(JSON.stringify(character));
24          });
25      } catch (e) {
26          res.status(500).send('error ' + e);
27      }
28  }
```

it creates an instance

it saves it

it sends response to client

The University Of Sheffield.

Questions?

Let’s have break now

Socket.io and WebRTC

Prof. Fabio Ciravegna
The University of Sheffield
f.ciravegna@shef.ac.uk

The University Of Sheffield.

Towards a different client/ server connection

socket.io

Traditional Client/Server architectures

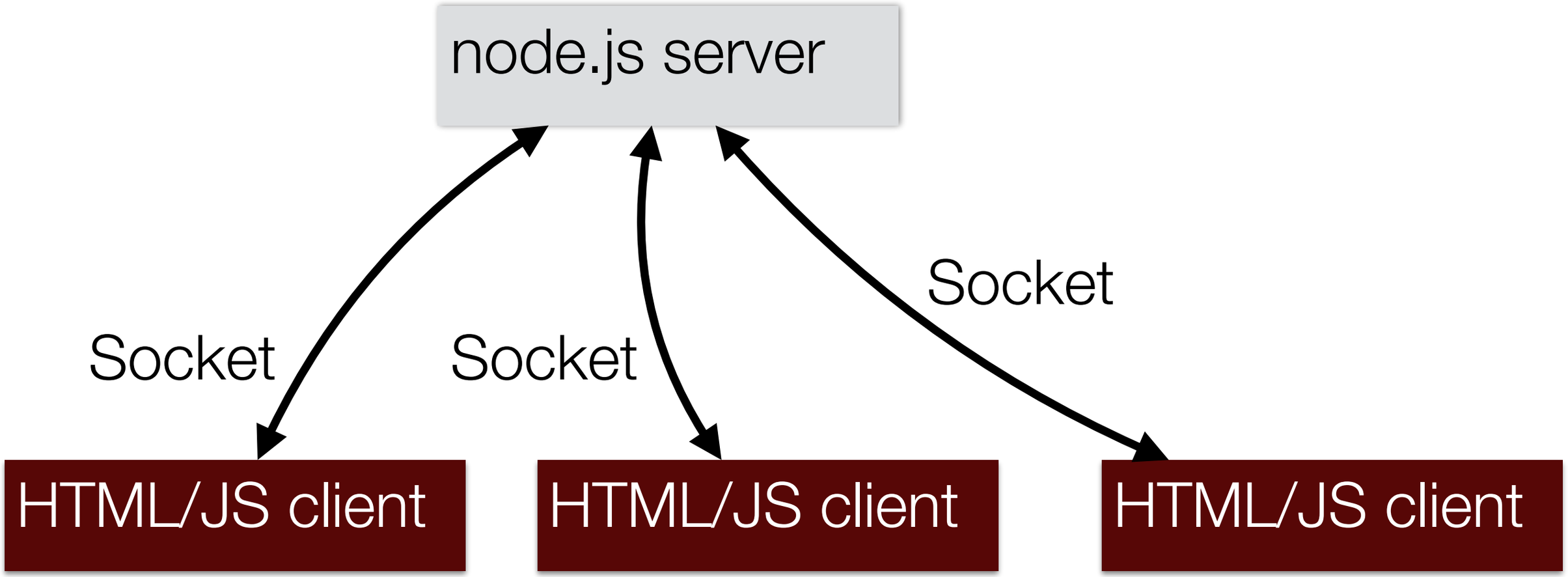
- When you make a request with Ajax/HTML/ Javascript you wait for the server to return results
- However in many cases
  - (e.g. twitter streaming API)
- you just would like the server to be able to reopen the connection and send more data
  - otherwise you need a client regularly polling the server to check if there are new results
    - rather cumbersome

Socket.io

Socket.io

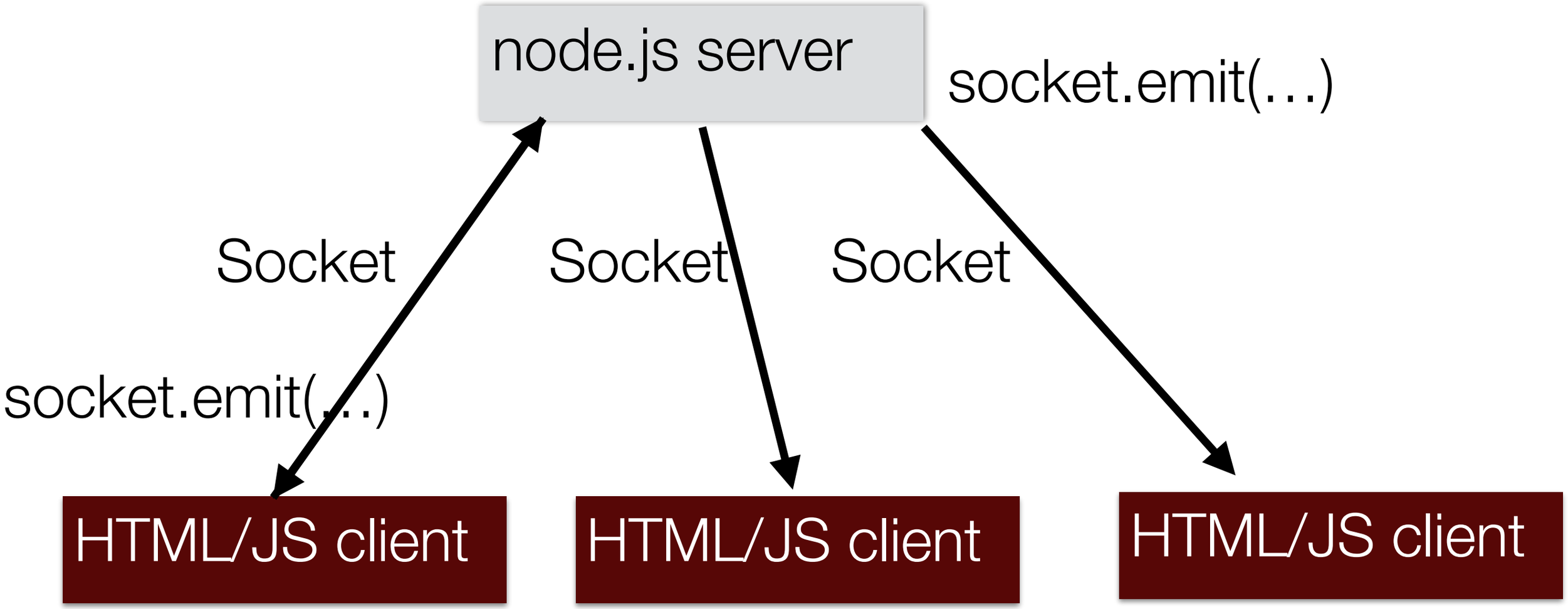
- Socket.IO enables real-time bidirectional event-based communication.
  - It works on every platform, browser or device, focusing equally on reliability and speed
  - It has two parts:
    - a client-side library that runs in the browser,
    - a server-side library for node.js.
  - Both components have a nearly identical API
- It primarily uses the WebSocket protocol
- It is possible to send any data,
  - Including blobs, i.e. Image, audio, video
- it is event based (on…)
- communication can be started by both client and server once connection is established and until it is closed from either sides

1 server, n clients, n sockets



- Socket is private channel shared by 1 client and 1 server
- However clients can communicate via the server

1 server, n clients, n sockets



- Communication happens via the command socket.emit(…) on both sides

Client Server communication

node.js server

```
var io = require('socket.io')(http);
file.serve(…);

io.on('connection', function(socket){
  socket.on ('message',
      function (param){…});
                  …
        socket.emit ('message' param)

});
```

HTML/JS client

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

  socket.emit ('message' param)

  socket.on ('message', function (param){…})
```

client must open the socket

socket automatically closes when client navigates away from page

The University Of Sheffield.

socket.io & Express

Using with Express 3/4

Server (app.js)

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

server.listen(80);

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

io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data
) {
    console.log(data);
  });
});
```

Client (index.html)

```
<script src="/socket.io/socket.io.js"></scri
pt>
<script>
  var socket = io.connect('http://localhost'
);
  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'dat
a' });
  });
</script>
```

The University Of Sheffield.

Sending and receiving events

Socket.IO allows you to emit and receive custom events. Besides `connect`, `message` and `disconnect`, you can emit custom events:

Server

```
// note, io(<port>) will create a http server for you
var io = require('socket.io')(80);

io.on('connection', function (socket) {
  io.emit('this', { will: 'be received by everyone'});

  socket.on('private message', function (from, msg) {
    console.log('I received a private message by ', from, ' saying ', msg);
  });

  socket.on('disconnect', function () {
    io.emit('user disconnected');
  });
});
```

Restricting yourself to a namespace

If you have control over all the messages and events emitted for a particular application, using the default / namespace works. If you want to leverage 3rd-party code, or produce code to share with others, socket.io provides a way of namespacing a socket.

This has the benefit of `multiplexing` a single connection. Instead of socket.io using two `WebSocket` connections, it'll use one.

We have two namespaces here: /chat and /news

Server (app.js)

```
var io = require('socket.io')(80);
var chat = io
  .of('/chat')
  .on('connection', function (socket) {
    socket.emit('a message', {
        that: 'only'
      , '/chat': 'will get'
    });
    chat.emit('a message', {
        everyone: 'in'
      , '/chat': 'will get'
    });
  });

var news = io
  .of('/news')
  .on('connection', function (socket) {
    socket.emit('item', { news: 'item' });
  });
```

Client (index.html)

```
<script>
  var chat = io.connect('http://localhost/ch
at')
    , news = io.connect('http://localhost/ne
ws');

  chat.on('connect', function () {
    chat.emit('hi!');
  });

  news.on('news', function () {
    news.emit('woot');
  });
</script>
```

The client refers to the channels via URLs server/channel

socket.io callbacks

Sending and getting data (acknowledgements)

Sometimes, you might want to get a callback when the client confirmed the message reception.

To do this, simply pass a function as the last parameter of `.send` or `.emit`. What's more, when you use `.emit`, the acknowledgement is done by you, which means you can also pass data along:

Server (app.js)

```
var io = require('socket.io')(80);

io.on('connection', function (socket) {
  socket.on('ferret', function (name, fn) {
    fn('woot');
  });
});
```

Client (index.html)

```
<script>
  var socket = io(); // TIP: io() with no ar
gs does auto-discovery
  socket.on('connect', function () { // TIP:
 you can avoid listening on `connect` and li
sten on events directly too!
    socket.emit('ferret', 'tobi', function (
data) {
      console.log(data); // data will be 'wo
ot'
    });
  });
</script>
```

DO NOT return a private message via socket.emit - it would be a public message!!!!

Broadcasting

- Broadcasting means sending a message to everyone else
  - except for the socket that starts it

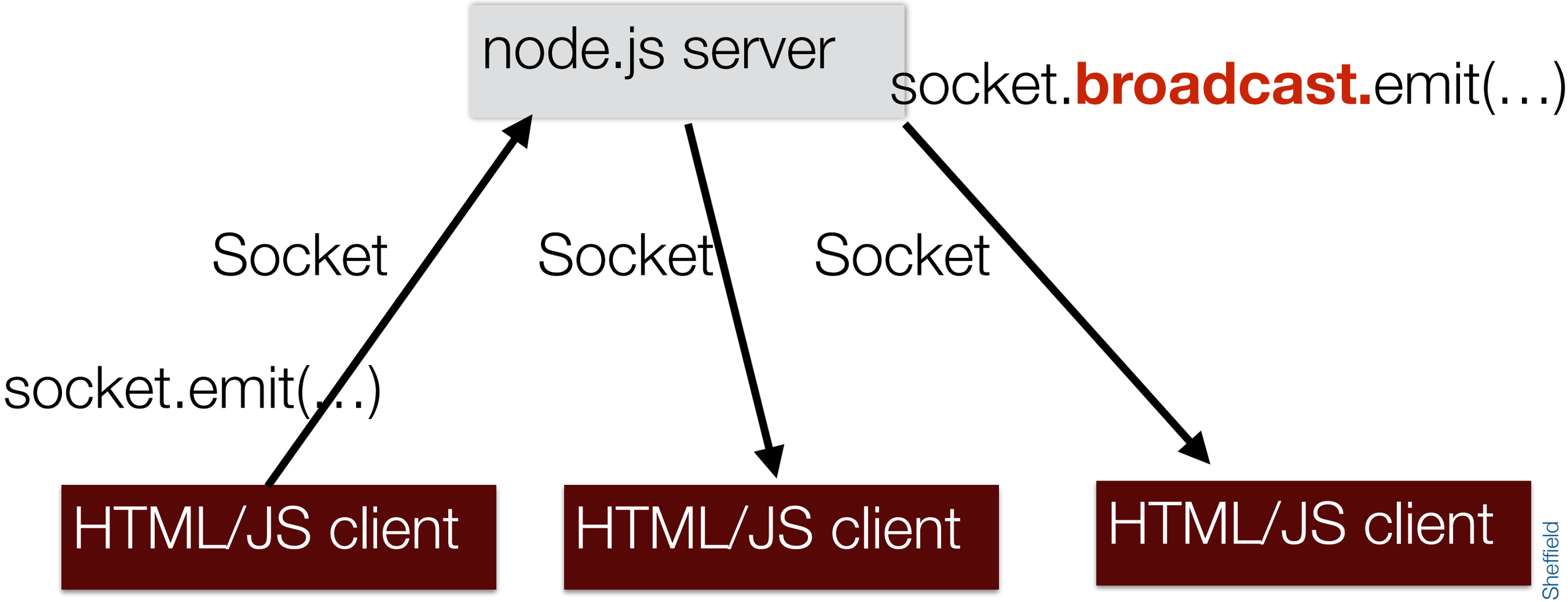
Broadcasting messages

To broadcast, simply add a `broadcast` flag to `emit` and `send` method calls. Broadcasting means sending a message to everyone else except for the socket that starts it.

Server

```
var io = require('socket.io')(80);

io.on('connection', function (socket) {
  socket.broadcast.emit('user connected');
});
```

socket.broadcast.emit



- Communication is not returned to the originating client

Rooms

Within each namespace, you can also define arbitrary channels that sockets can `join` and `leave`.

Joining and leaving

You can call `join` to subscribe the socket to a given channel:

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

And then simply use `to` or `in` (they are the same) when broadcasting or emitting:

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

This is on the server side
The client can be in just one room at a time

To leave a channel you call `leave` in the same fashion as `join`.

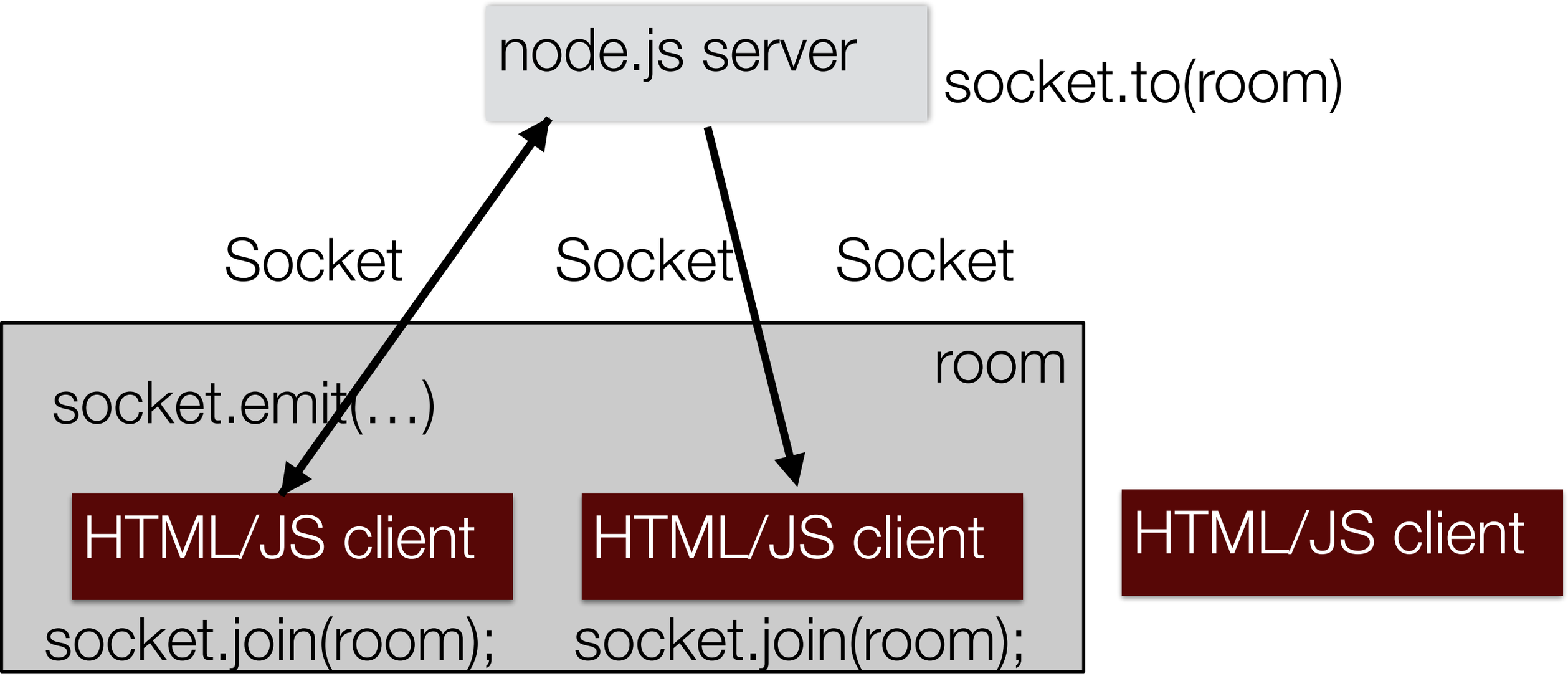
Default room

Each `Socket` in Socket.IO is identified by a random, unguessable, unique identifier `Socket#id`. For your convenience, each socket automatically joins a room identified by this id.

This makes it easy to broadcast messages to other sockets:

```
io.on('connection', function(socket){
  socket.on('say to someone', function(id, msg){
    socket.broadcast.to(id).emit('my message', msg);
  });
});
```

1 server, n clients, n sockets



- Once you are in a room, socket.emit(…) just reaches those in the same room

Disconnection

- e.g. when client moves away from page

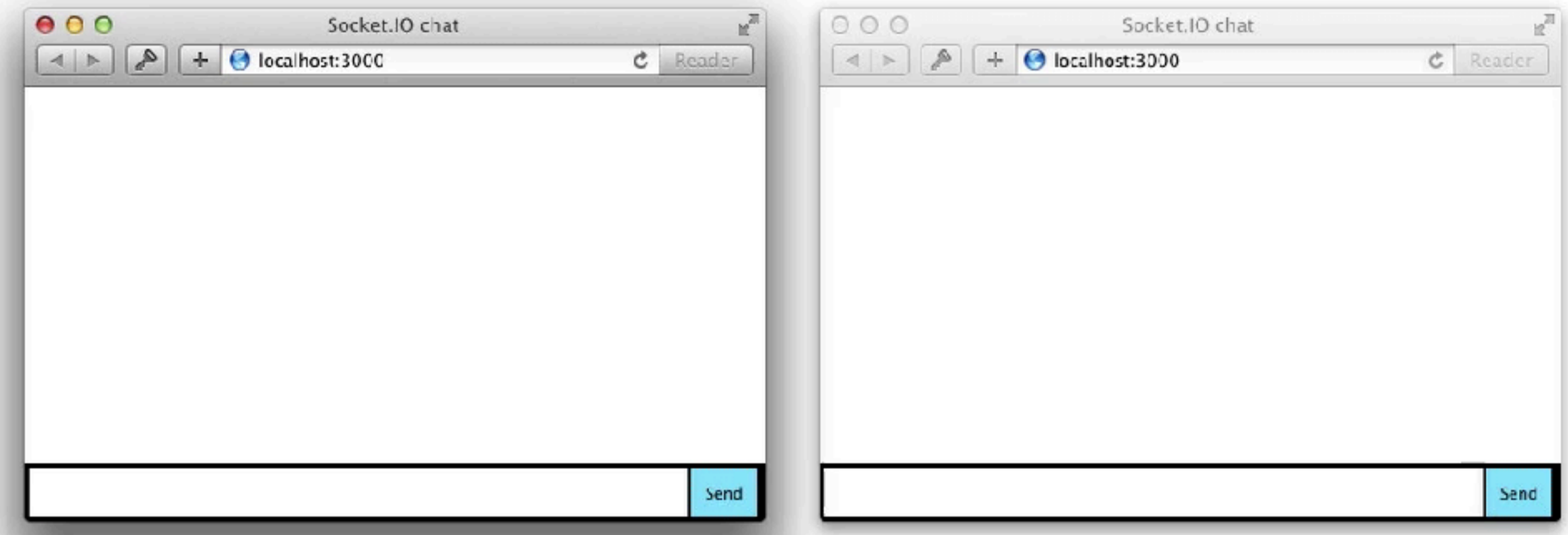
```
io.on('connection', function(socket){
  console.log('a user connected');
  socket.on('disconnect', function(){
    console.log('user disconnected');
  });
});
```

https://socket.io/get-started/chat/

The University Of Sheffield.

Instant messaging and chat

Whatsapp or Skype - like



Goal

- Creating an instant messaging and chat system
- Design:
  - Node.js/Express serves a file index.html
  - Index.html opens a socket and joins a room
    - the client tells the server it is joining a room
    - the server opens the room if not existing and joins the client to it
    - the server tells everybody in the room the client has joined
- every time the user writes and sends a message
    - the client sends the text to the server
    - the client writes on its own message panel
    - the server broadcasts it to everybody else
    - the other clients in the room write the message onto their message panels

joining a room

node.js server

```
file.serve()

io.on('connection', function(socket){
  socket.on ('joining',
        function (userId, roomId){
          socket.join(room);
          socket.to(room).emit
                ('updatechat',
                socket.username +
                ' has joined this room', '');});
});
```

socket.to(socket.room) broadcasts to all sockets in the room but the calling one

HTML/JS client 1

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

  socket.emit('joining', userId, roomId);
...write on message panel
```

HTML/JS client 2

```
<script src="/socket.io/socket.io.js">
</script>
...
<script>
  var socket = io();
  socket.on ('updatechat',
        function (message){
          ...write on message panel
        });
```

sending a message

node.js server

```
io.on('connection', function(socket){
  socket.on ('joining',
        function (userId, roomId){
          socket.join(room);
          socket.broadcast.to(room).emit
                ('updatechat',
                socket.username +
                ' has joined this room', '');});});
socket.on('sendchat', function (data) {
    io.sockets.in(socket.room).emit
        ('updatechat', socket.username,
        data);
 });});
```

io.sockets.in broadcasts to all sockets in the room (if you use io.sockets.in the sender does not need to write independently onto its own panel)

HTML/JS client 1

```
socket.emit('sendchat', message);

socket.on ('updatechat',
        function (message){
          …write on message panel
        });
```

HTML/JS client 2

```
<script src="/socket.io/socket.io.js">
</script>
…
<script>
  var socket = io();
  socket.on ('updatechat',
        function (message){
          …write on message panel
        });
```

The University Of Sheffield.

The rest is just a form!

You are in room: 3946

User 1494 has joined this room:

User 1494: hello!

me: hello to you!

User 1494: it is good to see you

me: indeed!

Anyway!

Send

The University Of Sheffield.

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>My Chat App</title>
    <link href="../css/style.css" rel="stylesheet">
</head>
<body onload="initialiseUserAndRoom();">
<script src="/socket.io/socket.io.js"></script>
<script src="../js/main.js"></script>

<h1 id="roomNo"></h1>
<div>
    <div id="chat"></div>
    <form action="" onsubmit="return sendText()">
        <input id="text" autocomplete="off" autofocus/>
        <button>Send</button>
    </form>


</div>
</body>
```

The University Of Sheffield.

```
/**
 * It sends a message when the user presses the button or return
 * @returns {boolean}
 */
function sendText() {
    var inpt = document.getElementById('text');
    var text = inpt.value;
    if (text == '')
        return false;
    socket.emit('sendchat', text);
    inpt.value = '';
    return false;
}
```

```
var inColour = false;
var previousWriter = '';
var roomId;
var userId;


var socket = io();
socket.on('updatechat', function (who, text) {
    var divI = document.getElementById('chat');
    var dv2 = document.createElement('div');
    divI.appendChild(dv2);
    dv2.style.backgroundColor = getChatColor(who);
    var whoisit = (who == userId) ? 'me' : who;
    dv2.innerHTML = '<br/>'+whoisit + ': ' + text+'<br/><br/>';
});

/**
 * it initialises the user and the room
 * here you should include the login/password request
 */
function initialiseUserAndRoom() {
    var rndmId = 'User ' + Math.floor((Math.random() * 10000) + 1);
    userId = prompt("Please enter your name", rndmId);
    // if cancel is selected
    if (userId == null) userId = rndmId;

    var randomRoomId = Math.floor((Math.random() * 10000) + 1);
    roomId = prompt("What room would you like to join?", randomRoomId);
    // if cancel is selected
    if (roomId == null) roomId = randomRoomId;
    socket.emit('joining', userId, roomId);
    document.getElementById('roomNo').innerHTML= 'You are in room: '+roomId;
}
```

WebRTC

WebRTC

- WebRTC (Web Real-Time Communication) is
  - an API definition drafted by the World Wide Web Consortium (W3C)
  - that supports browser-to-browser applications
    - for voice calling, video chat, and P2P file sharing
    - **without the need of either internal or external plugins**
    - WebRTC is a free, open project
- This means that:
  - With WebRTC it is possible to create a Skype-like application that works in a browser
- WebRTC is made possible by the availability of bidirectional channels like socket.io

WebRTC

https://tokbox.com/about-webrtc

- WebRTC is made up of three APIs:
  - GetUserMedia
    - Camera, microphone, screen, etc. access
  - PeerConnection
    - Sending and receiving media
  - DataChannels
    - sending non-media direct between browsers
- The development of WebRTC is supported by the W3C, Google, Mozilla, and Opera
  - currently supported in Opera, Google Chrome versions 23+, and Mozilla Firefox versions 22+, Safari from 11.1

The University Of Sheffield.

Why is WebRTC important?

The WebRTC project is incredibly important as it marks the first time that a powerful real-time communications (RTC) standard has been open sourced for public consumption. It opens the door for a new wave of RTC web applications that will change the way we communicate today.

Significantly better video quality	WebRTC video quality is noticeably better than Flash.
Up to 6x faster connection times	Using JavaScript WebSockets, also an HTML5 standard, improves session connection times and accelerates delivery of other OpenTok events.
Reduced audio/video latency	WebRTC offers significant improvements in latency through WebRTC, enabling more natural and effortless conversations.
Freedom from Flash	With WebRTC and JavaScript WebSockets, you no longer need to rely on Flash for browser-based RTC.
Native HTML5 elements	Customize the look and feel and work with video like you would any other element on a web page with the new video tag in HTML5.

GetUserMedia

http://www.html5rocks.com/en/tutorials/webrtc/basics/

- navigator.getUserMedia()
  - Webcam and microphone input are accessed without a plugin.
- Checking if browser supports it

```
function hasGetUserMedia() {
  // !! converts a value to a boolean and ensures a boolean type.
  return !!(navigator.getUserMedia ||
            navigator.webkitGetUserMedia ||
            navigator.mozGetUserMedia ||
            navigator.msGetUserMedia);}


if (hasGetUserMedia()) {
  // Good to go!
} else {
  alert('getUserMedia() is not supported in your browser'
}
```

GetUserMedia

- The first parameter to getUserMedia() is an object specifying the details and requirements for each type of media you want to access.
- For example {video: true, audio: true} will access both video and audio

```
var session = {
    audio: true,
    video: true
    },
};
```

- (remember: do not use <scripts> tags in html files. Use js files!)

```
<video autoplay></video>

<script>
  var errorCallback = function(e) {
    console.log('Rejected!', e);
  };

  //
  navigator.getUserMedia({video: true, audio: true},
     function(localMediaStream) {
       var video = document.querySelector('video');
       video.src =
          window.URL.createObjectURL(localMediaStream);

 }, errorCallback);
</script>
```

HTML5 container for vide

callback function returns a videostream

window.URL.createObjectURL(). This method creates a simple URL string which can be used to reference data stored in a Blob object

Video parameters

- Video and audio can have parameters
  - Instead of just indicating basic access to video
    - e.g. {video: true})
  - You can additionally require the stream to be HD

```
var hdConstraints = {
  video: {
    mandatory: {
      minWidth: 1280,
      minHeight: 720
    }
  }
};

navigator.getUserMedia(hdConstraints, successCallback,
errorCallback);
```

Choosing the camera

```
cameras = [];
function getSources(sourceInfos) {
    var fillCameras = 0;
    for (var i = 0; i !== sourceInfos.length; ++i) {
        var sourceInfo = sourceInfos[i];
        if (sourceInfo.kind === 'video') {
            var text = sourceInfo.label ||
                    'camera ' + (cameras.length + 1);
            cameraNames[fillCameras] = text;
            cameras[fillCameras++] = sourceInfo.id;
        } else if (sourceInfo.kind === 'audio') {
            audioSource = sourceInfo.id;
        }    }
    videoSource = cameras[cameras.length - 1];
}

MediaStreamTrack.getSources(getSources);
```

In phones the first camera is the front facing one.
The last camera is the back camera

choosing (2)

```
function sourceSelected(audioSource, videoSource) {
  var constraints = {
    audio: {
      optional: [{sourceId: audioSource}]
    },
    video: {
      optional: [{sourceId: videoSource}]
    }
};

navigator.getUserMedia(constraints,
        successCallback, errorCallback);
```

Taking snapshot via canvas

```
<video autoplay></video>
<img src="">
<canvas style="display:none;"></canvas>
<script>
  var video = document.querySelector('video');
  var canvas = document.querySelector('canvas');
  var ctx = canvas.getContext('2d');
  var localMediaStream = null;
  video.addEventListener('click', snapshot, false);
  navigator.getUserMedia({video: true}, function(stream) {
    video.src = window.URL.createObjectURL(stream);
    localMediaStream = stream;
  }, errorCallback);

  function snapshot() {
    if (localMediaStream) {
      ctx.drawImage(video, 0, 0);
      document.querySelector('img').src
        = canvas.toDataURL('image/png');
    }  }

</script>
```

event click on video

saving the local stream

creating png image
from localMediaStream

Sending to a server

```
sendImage(userId, canvas.toDataURL());
```

```
function sendImage(userId, imageBlob) {
    var data = {userId: userId, imageBlob: imageBlob};
     $.ajax({
         dataType: "json",
         url: '/uploadpicture_app',
         type: "POST",
         data: data,
         success: function (data) {
             token = data.token;
             // go to next picture taking
             location.reload();
         },
         error: function (err) {
             alert('Error: ' + err.status + ':' + err.statusText);
         }
});
```

Server side

```
router.post('/uploadpicture_app',
    function (req, res) {
      var userId= req.body.userId;
       var newString = new Date().getTime();
       targetDirectory = './private/images/' + userId + '/';
        if (!fs.existsSync(targetDirectory)) {
            fs.mkdirSync(targetDirectory);
        }
        console.log('saving file ' + targetDirectory + newString);

        // strip off the data: url prefix to get just the base64-encoded bytes
        var imageBlob = req.body.imageBlob.replace(/^data:image\/\w+;base64,/, "");
        var buf = new Buffer(imageBlob, 'base64');
        fs.writeFile(targetDirectory + newString + '.png', buf);

        var filePath = targetDirectory + newString;
        console.log('file saved!');

        var data = {user: userId, filePath: filePath};
       var errX = pictureDB.insertImage(data);
        if (errX) {
            console.log('error in saving data: ' + err);
            return res.status(500).send(err);
        } else {
            console.log('image inserted into db');
        }
        res.end(JSON.stringify({data: ''}));
        });
});
```

The University Of Sheffield.

Applying effects

```
<style>
video { background: rgba(255,255,255,0.5); border: 1px solid #ccc;}
.grayscale { +filter: grayscale(1);}
.sepia { +filter: sepia(1);}
.blur { +filter: blur(3px);}
</style>
<video autoplay></video>
<script>
var idx = 0;
var filters = ['grayscale', 'sepia', 'blur', 'brightness',
               'contrast', 'hue-rotate', 'hue-rotate2',
               'hue-rotate3', 'saturate', 'invert', ''];

function changeFilter(e) {
  var el = e.target;  el.className = '';
    // loop through filters.
  var effect = filters[idx++ % filters.length];
  if (effect) { el.classList.add(effect);  }}

document.querySelector('video').addEventListener(
    'click', changeFilter, false);
</script>
```

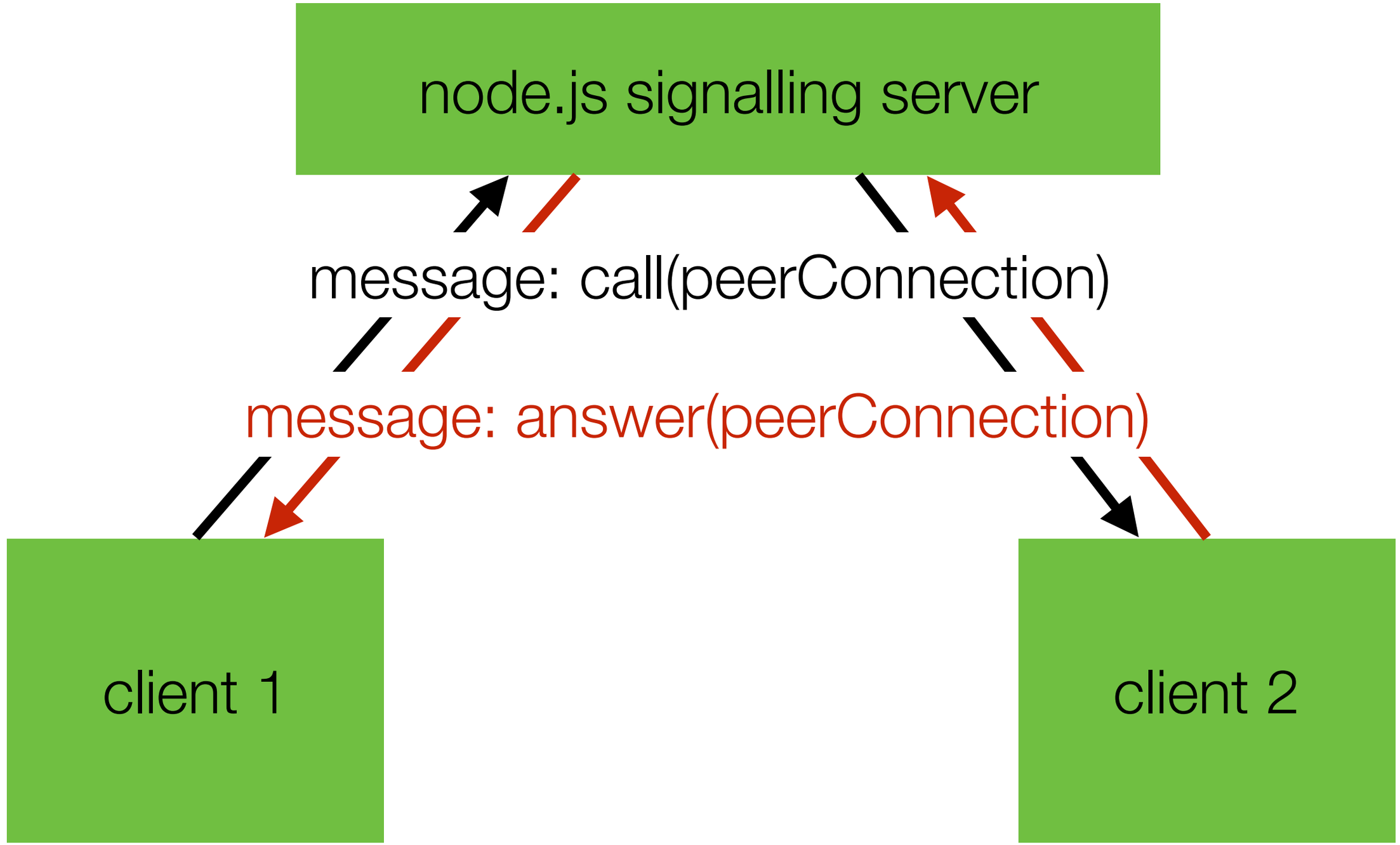
The University Of Sheffield.

PeerConnection

- It creates the connection between the two clients via a signalling server
  - the role of the signalling servers is to pass the messages between the two clients via socket.io

```
function createPeerConnection() {
  try {
   pc = new webkitRTCPeerConnection(
           { "iceServers":
             [{ "url": "stun:stun.l.google.com:19302" });
  pc.onicecandidate = handleIceCandidate;
                                  callback when remote stream is added
  pc.onaddstream = handleRemoteStreamAdded;
                                   callback when remote stream is removed
  pc.onremovestream = handleRemoteStreamRemoved;

} catch (e) {
     alert('Cannot create RTCPeerConnection object.');
    return;  }}
```

Calling via socket.io



The University Of Sheffield.

Calling

on the initiating partner

```
peerConnection.createOffer(callRemote);
function callRemote(description) {
  peerConnection.setLocalDescription(description);
  socket.emit('message', description)
```

on the server, send it to the partner…

```
socket.on('message', function (message) {
        socket.to(room).emit('message', message);
        });
```

on the remote client…

```
socket.on('message', function (message){
    if (message.type === 'offer') {
      if (!activeRTPConnection)
        peerConnection= createPeerConnection();
      peerConnection.createAnswer(answerRemote);
  });
function answerRemote(description) {
 peerConnection.setLocalDescription(description);
 socket.emit(‘message', description);}
```

Showing stream

- Receiving the connection description fires the event ‘pc.onaddstream’
  - Its callback function will receive as input the event that has caused the callback to be activated
  - the event has a stream field containing the userMedia stream (audio, video…) of the remote client
    - we create a blob and assign it as src of our HTML video element
    - now the remote stream is visible in our browser

```
function handleRemoteStreamAdded(event) {
  var remoteVideo= document.getElementById(‘remote_video’);
  remoteVideo.src = window.URL.createObjectURL(event.stream);
   remoteStream = event.stream;
}
```

Firewalls!

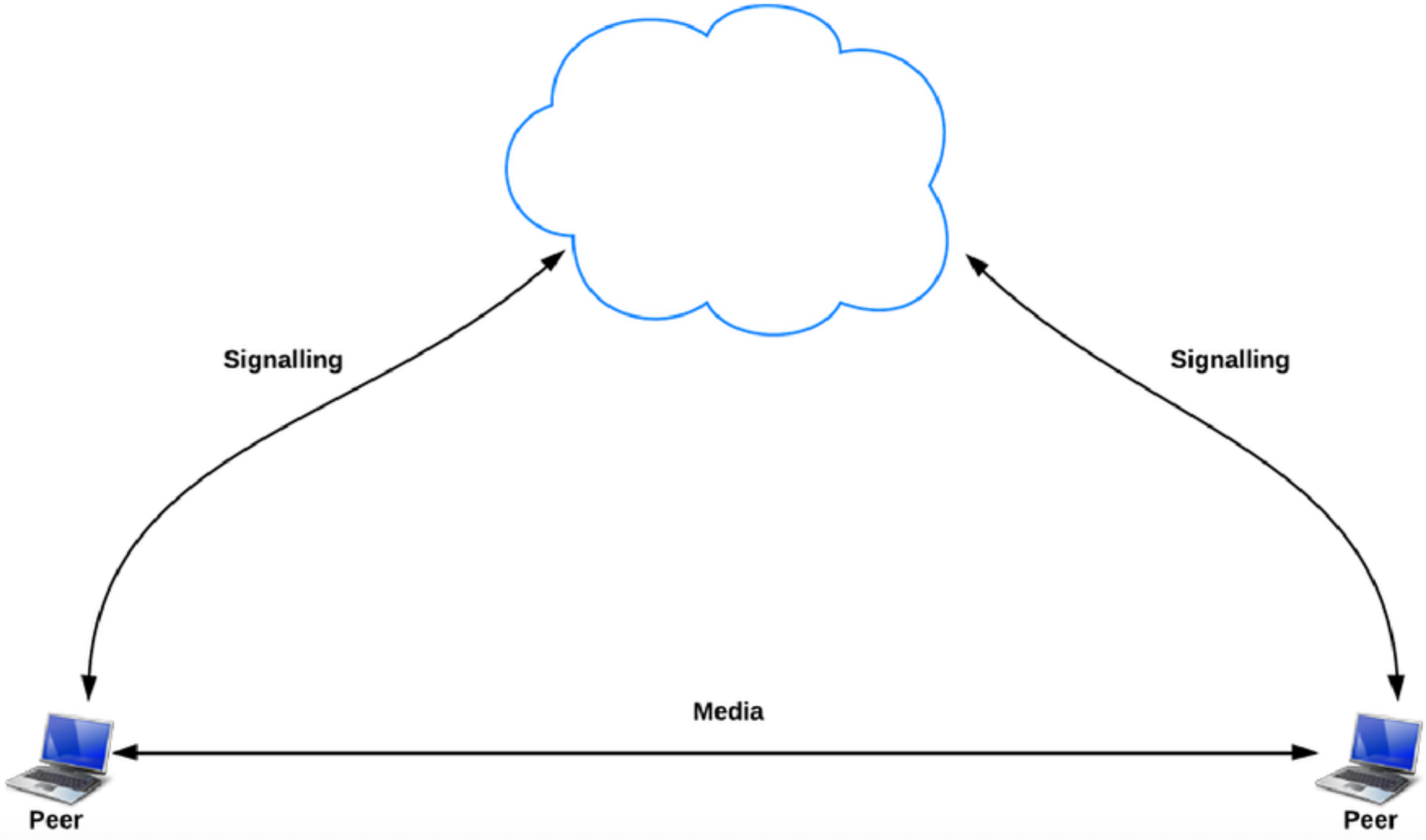
http://www.html5rocks.com/en/tutorials/webrtc/basics/#toc-signalling

- A realistic situation however prevents this from working because firewalls prevent clients to see each other directly
- WebRTC needs four types of server-side functionality:
  - User discovery and communication.
  - Signalling.
  - NAT/firewall traversal.
  - Relay servers in case peer-to-peer communication fails
- The STUN protocol and its extension TURN are used by the ICE framework to enable RTCPeerConnection to cope with NAT traversal

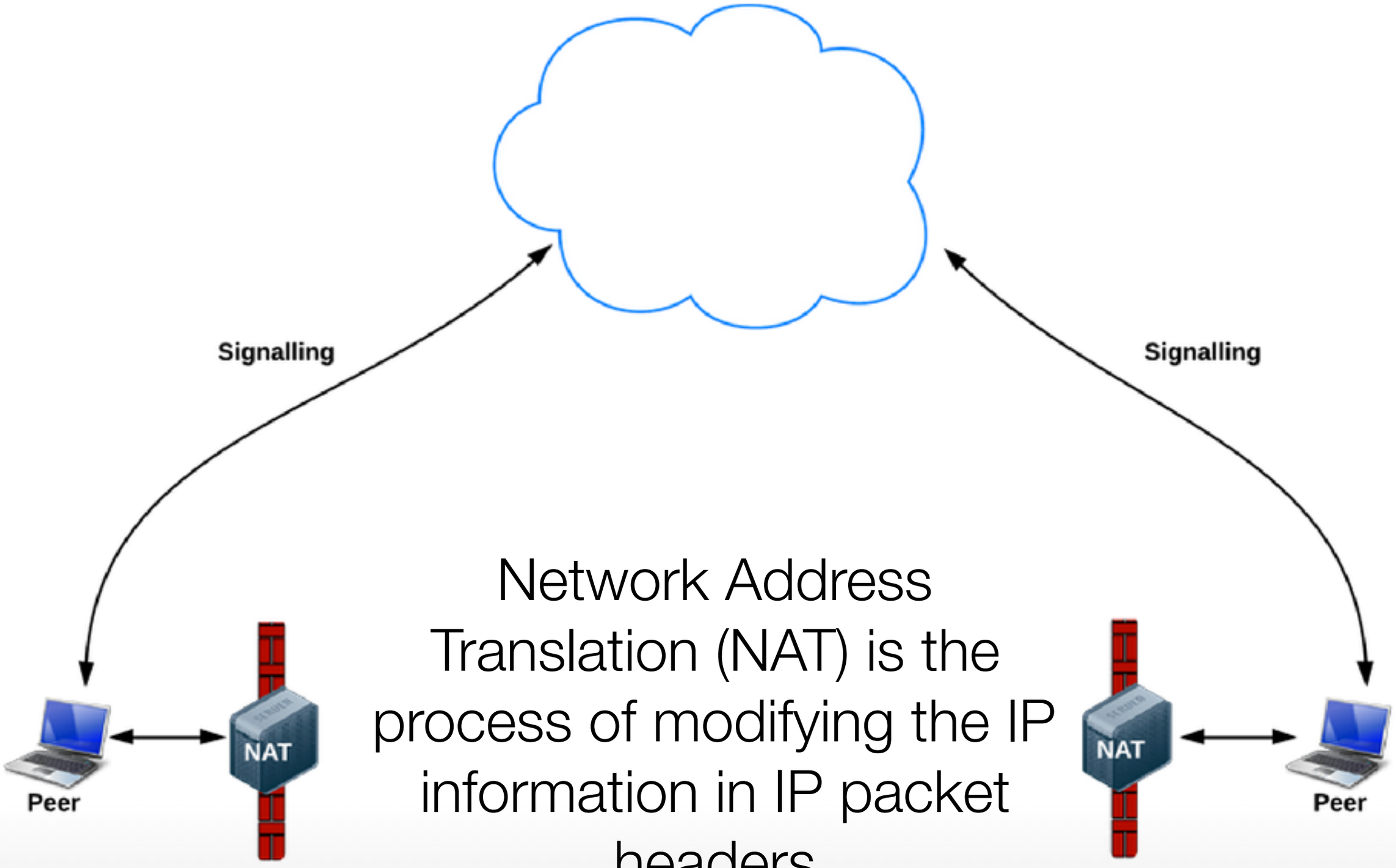
The University Of Sheffield.

An Ideal World

http://io13webrtc.appspot.com/#45



The Real World

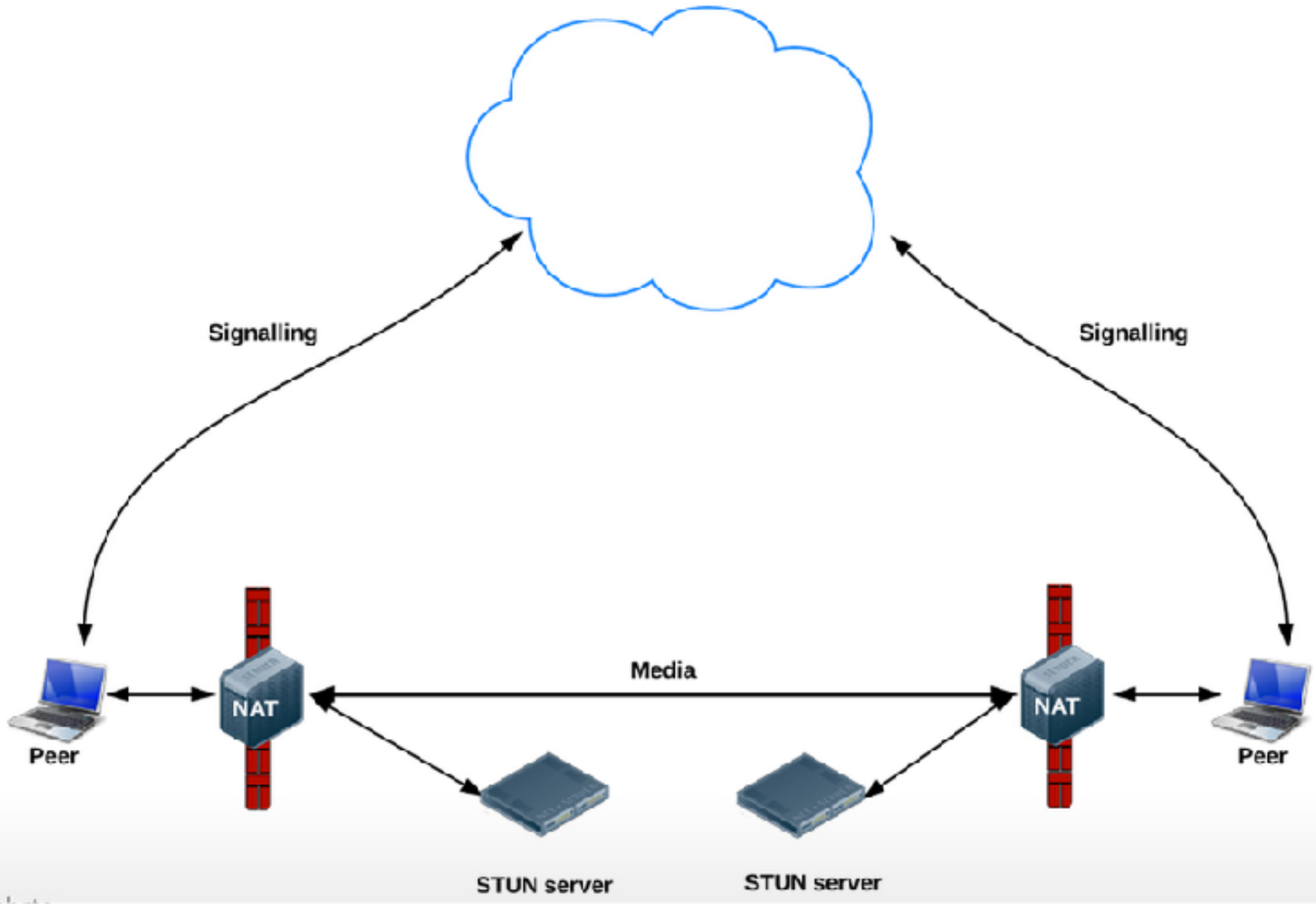


Network Address Translation (NAT) is the process of modifying the IP information in IP packet headers

https://www.bestvpn.com/blog/4246/what-is-a-nat-firewall/

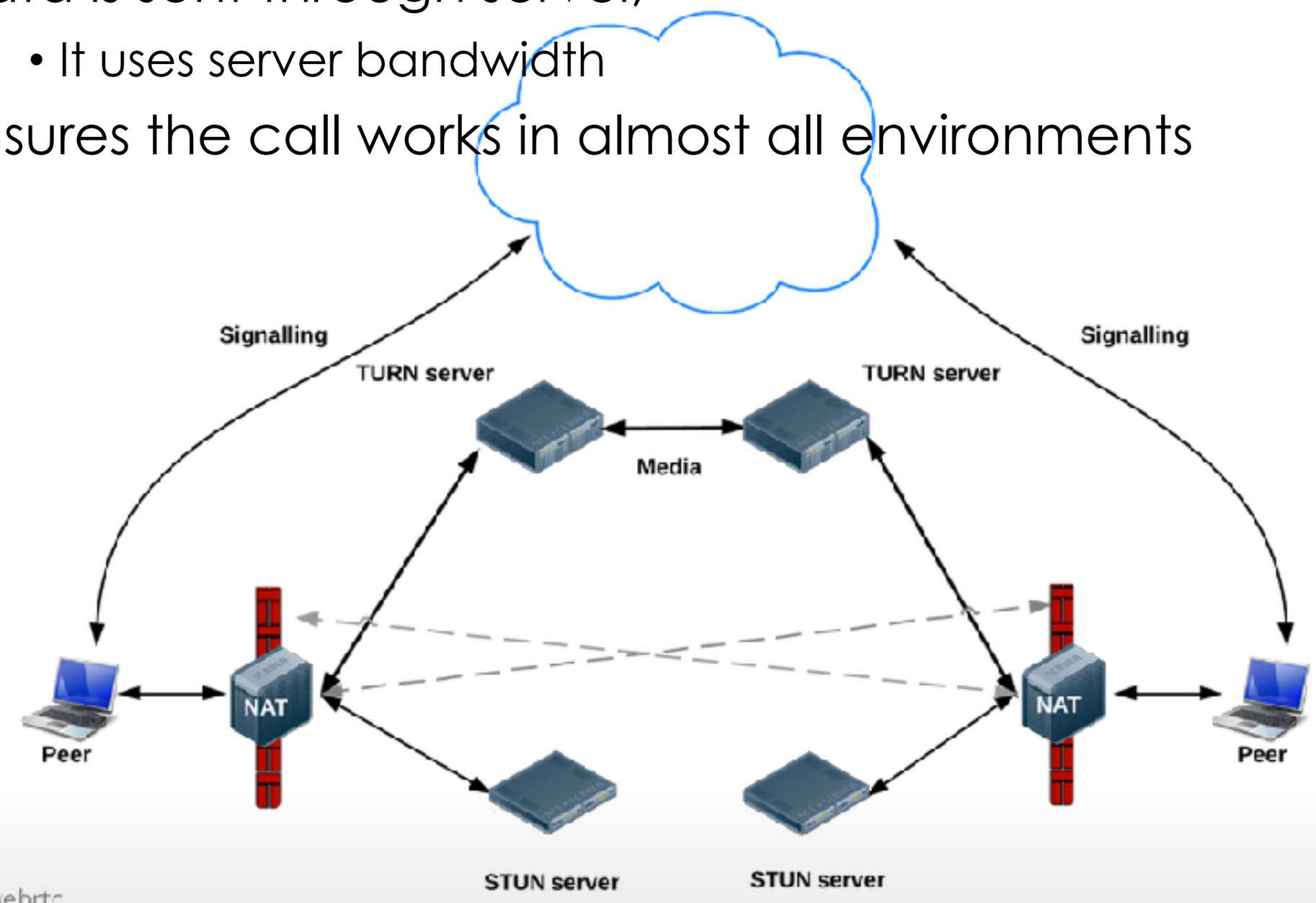
STUN Server

- Tell me what my public IP address is
  - Simple server, cheap to run
  - Data flows peer-to-peer



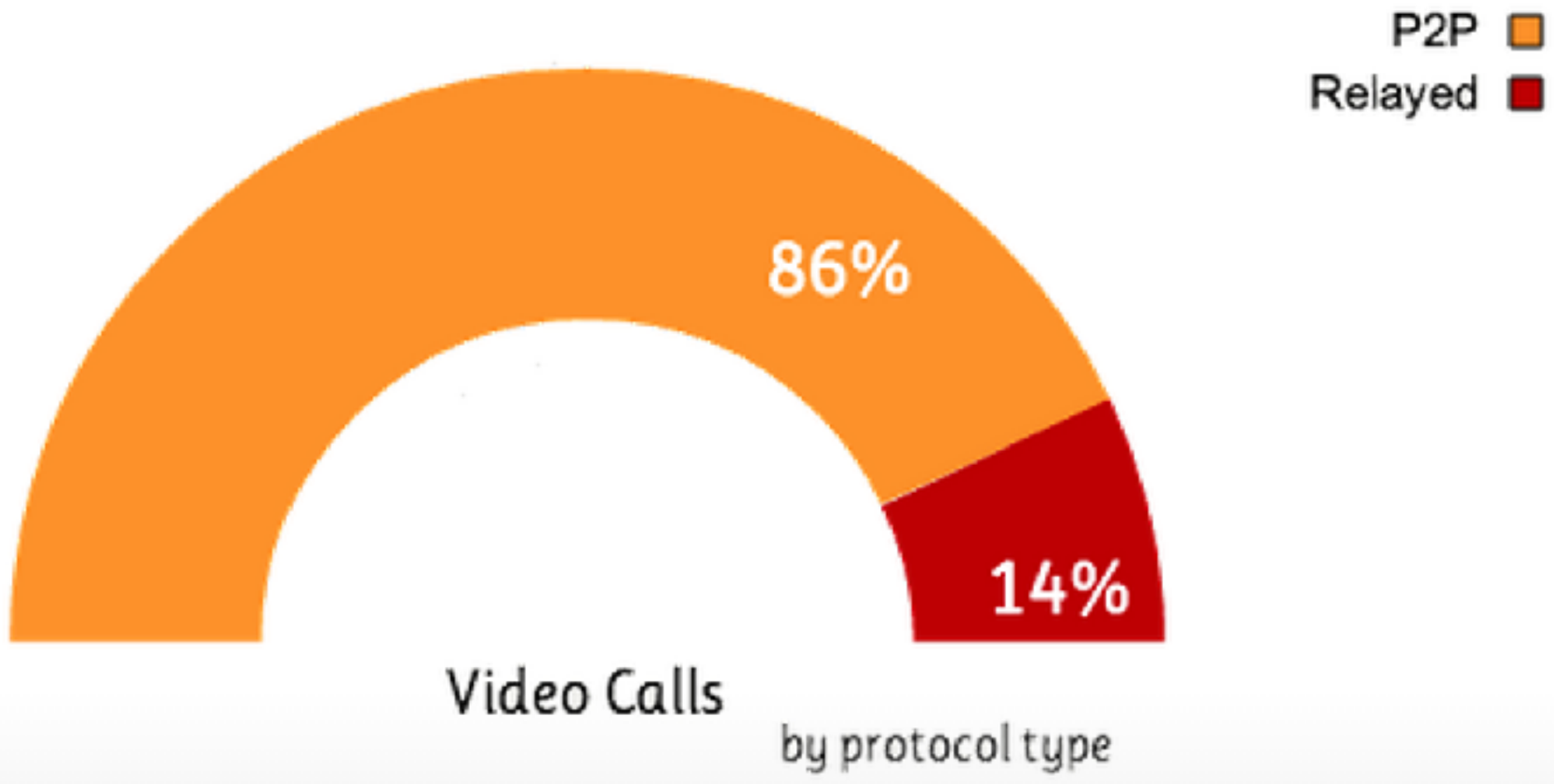
TURN Server

- Provide a cloud fallback if peer-to-peer communication fails
  - Data is sent through server,
    - It uses server bandwidth
  - Ensures the call works in almost all environments



ICE

- ICE: a framework for connecting peers
  - Tries to find the best path for each call
  - Vast majority of calls can use STUN (webrtcstats.com):



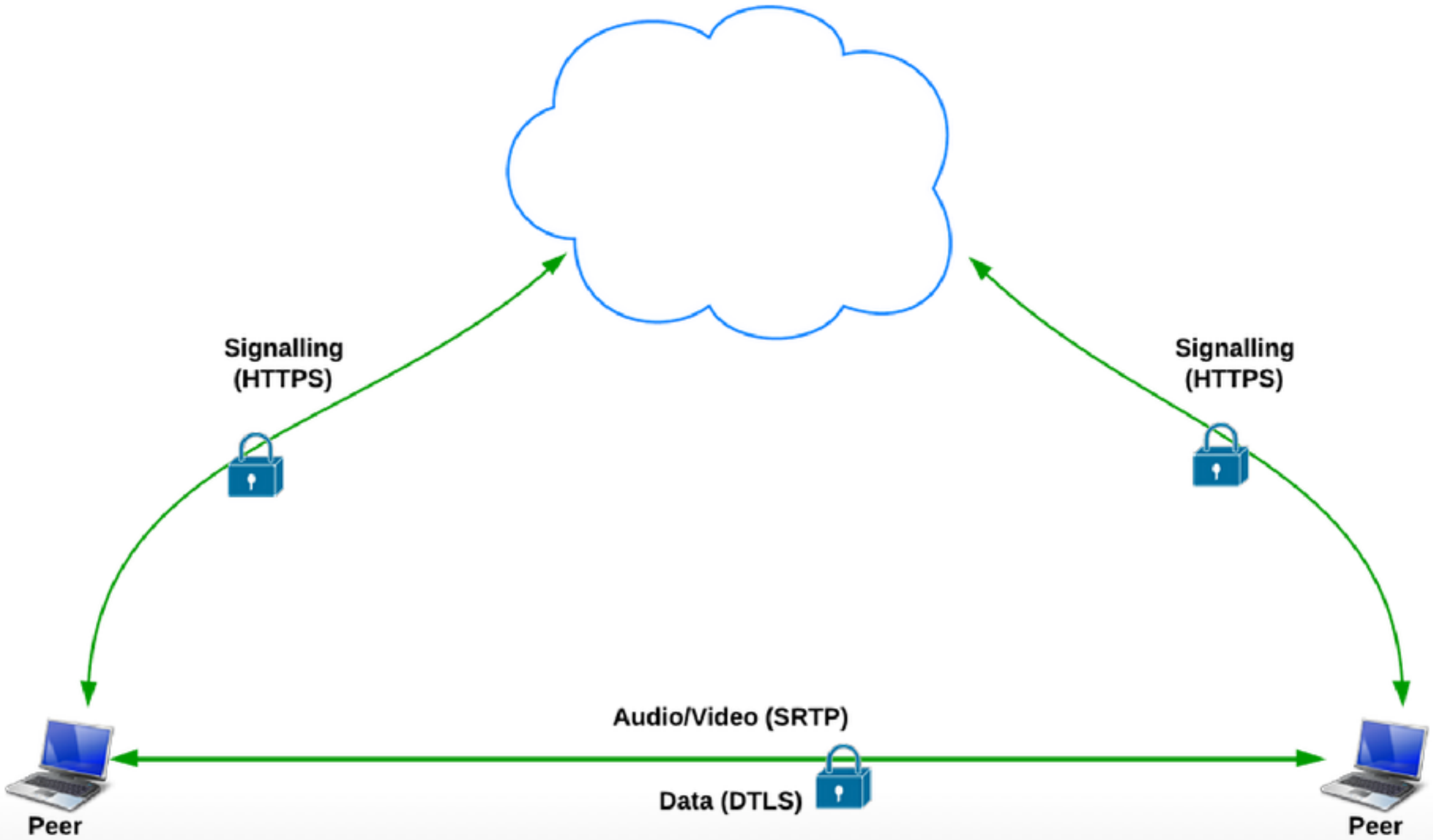
Declaring turn/stun servers

- In the RTCPeerConnection declaration

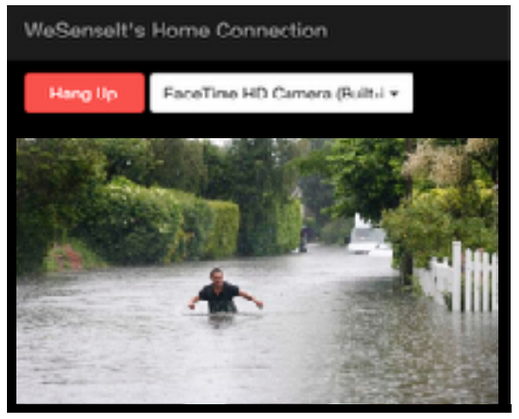
```
pc = new webkitRTCPeerConnection(
    { "iceServers":
       [
        { "url": "stun:stun.l.google.com:19302" },
        { "url":"turn:<<login>>@<<give TURN url>>>",
          "credential":<<password>>}],
          optional: [{RtpDataChannels: true}
       ]
    });
```

Always Use HTTPS

- it is just a small change in the node.js setting

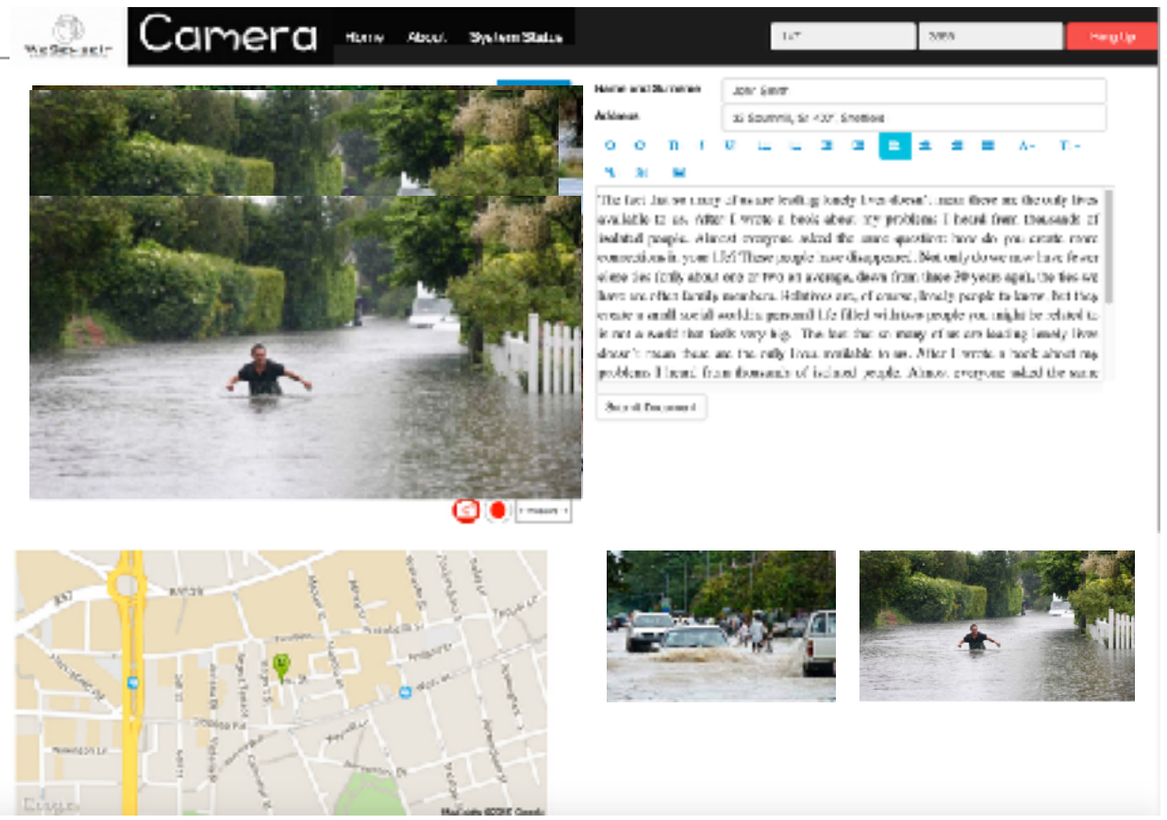


WeSenseIt
Citizen Observatories



WeSenseIt Camera

Prof Fabio Ciravegna
Department of Computer Science
The University of Sheffield



WSI Citizen Camera

- Citizen Camera is a cross-platform application that enables turning a citizen's camera into an street camera for the emergency control room.
  - Connecting with the citizen (via Android mobiles, IOS devices and personal computers running the Chrome Browser).
  - Seeing through their camera while simultaneously communicating with them via audio (so to e.g. give instructions or asking questions);
  - Recording the conversation
  - Recording the location of the caller
  - Taking screenshots
  - Taking notes (log). Log can be exported as independent document

Control Room

WeSenseIt
Citizen Water Observatories

Camera

Home About System Status

CR1743 R66973 Hang Up

Name and Surname John Smith

Address 33 Wandermall Road S12 3NG

Securing the society against disasters is one of the central elements of the functioning of any society. There is barely any societal sector which is not to some extent concerned by disasters and related resilience and security issues. The objective of this call is to reduce the loss of human life, environmental, economic and material damage from natural and man-made disasters, including extreme weather events, crime and terrorism threats.

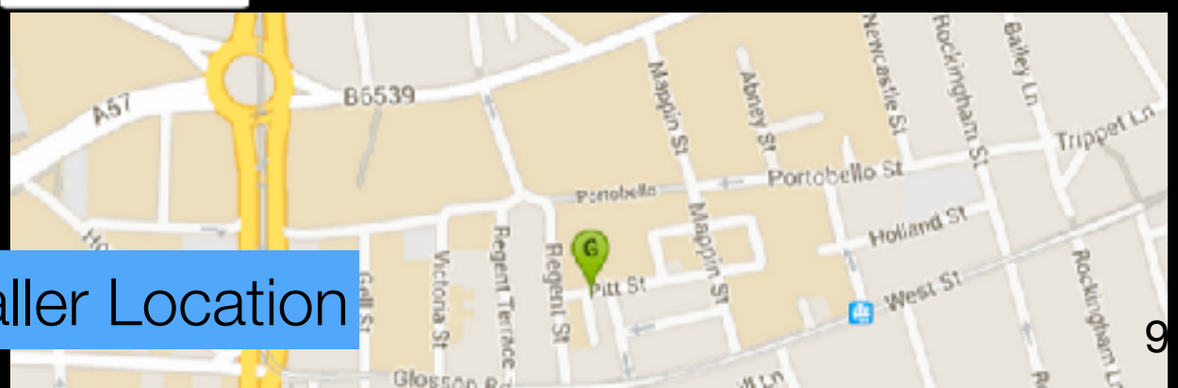
Notes

Submit Document

Time Connected: 00:00:25 You are connected

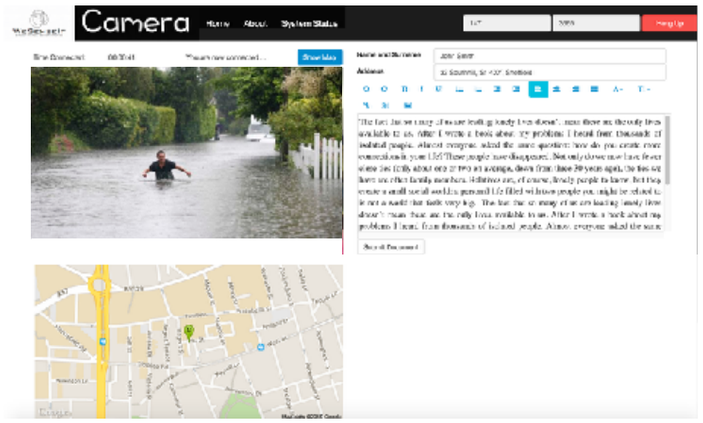
Live Stream

Recorded images and videos

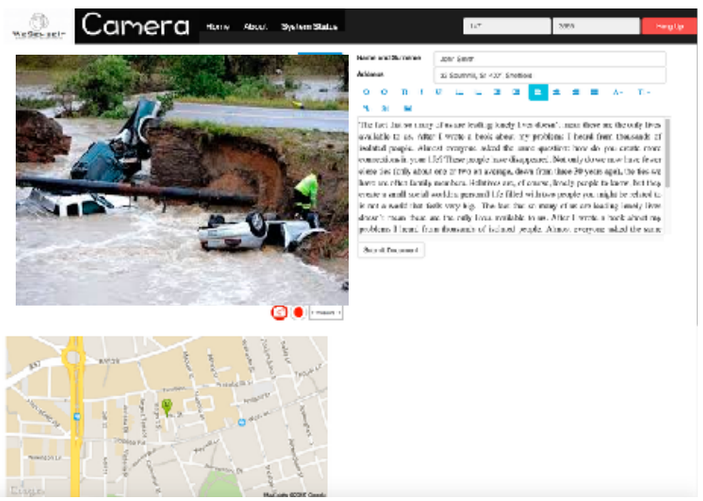


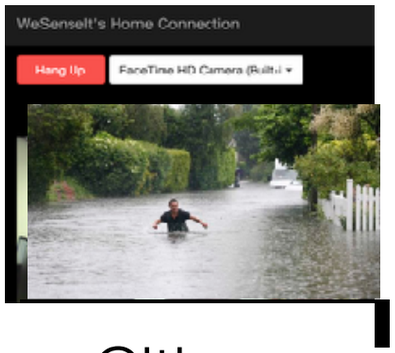
Caller Location

WeSenseIt
Citizen Water Observatories



Control Room



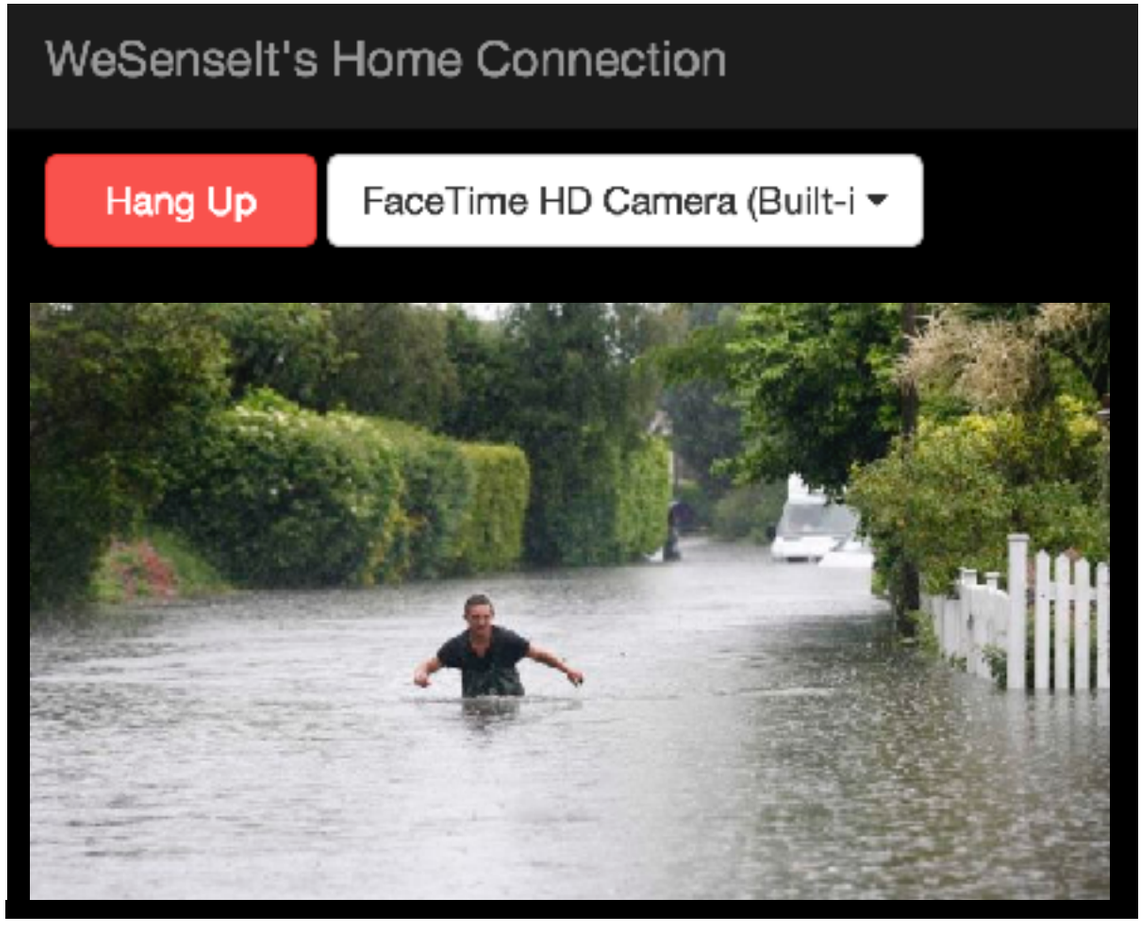


Citizen

Citizen

Citizen

Mobile Client

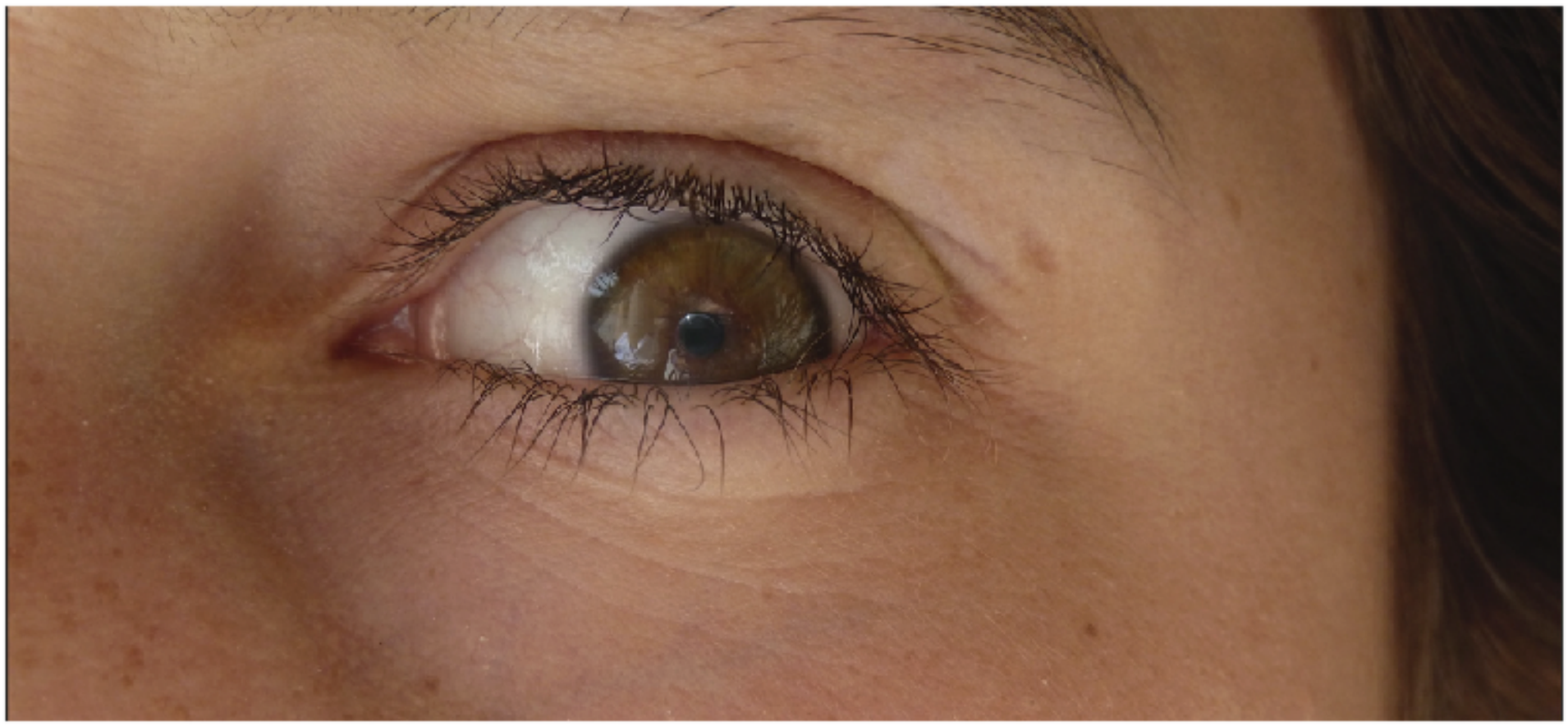


Exploitation: Sheffield Hospital

- Occupational Therapy
  - Send people home after accidents
  - Have to visit patients' houses
    - Often hundreds km away
  - Remote visit could save the NHS
    - Thousands of pounds in expenses
    - Thousands of bed days
- Project with Northern General Hospital

The University Of Sheffield.

With My Own Eyes | Home | About | System Status | Open Room | Master: presenter-88 | 100 | Connect

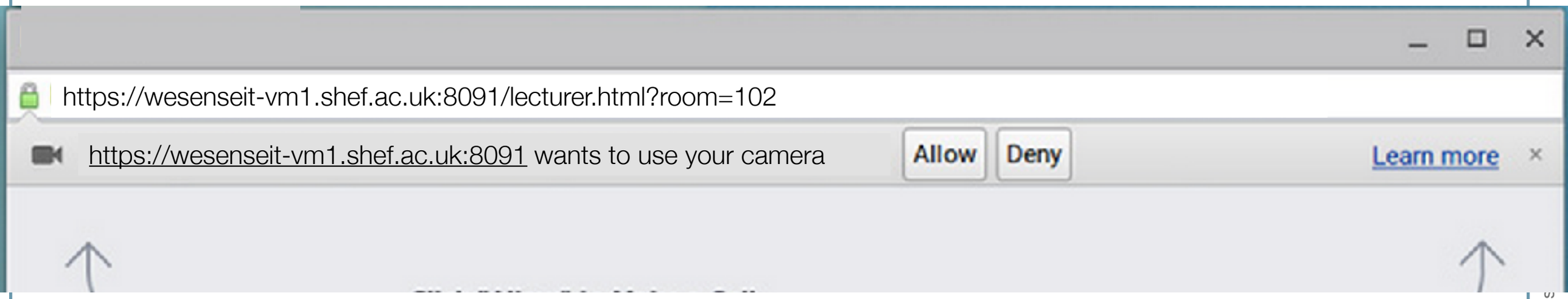


With My Own Eyes

Exploitation (2): With My Own Eyes

- A tool to support full participation of visually impaired students in lectures
- A large amount of information in student courses is delivered visually
  - A visually impaired student may therefore be at a significant disadvantage.
- Staff are recommended to provide reading material and slides in advance,
  - But presentations are becoming more and more multimedia and you cannot print a video or an animation.
  - Equally in lab classes demonstrations are run live
    - The inability to see seriously affects visually impaired students who are missing out a substantial learning opportunity

WeSenseIt
Citizen Water Observatories



MOE: Goal

- The goal of the project is to finalise and test in real life conditions
  - i.e. during lectures with visually impaired students
  - a tool which enables mirroring the screen of the lecturer into a student's laptop or tablet
  - As the mirroring happens in a normal browser, the student will be able to use any usual support tool, e.g. magnifiers.
  - The student will therefore be able to fully participate in the learning experience.

Exploitation (2): With My Own Eyes

- Students with visual impairment
- Speaker shares screen with students
- Already presented to 2 people with visual impairment
  - Said this could change their lives
- To be released as open source for free use in all schools and universities
- Alumni Association provided funds for 2 summer internship

The University Of Sheffield.

Questions?