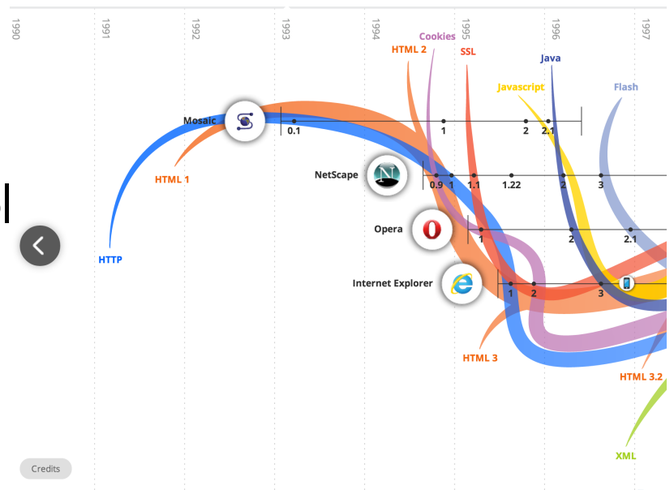
The Http Protocol

Professor Fabio Ciravegna
University of Sheffield



HTTP Protocol

- HTTP (for HyperText Transfer Protocol) is the primary method used to convey information on the World Wide Web http://en.wikipedia.org/wiki/Http_protocol
- HTTP is a protocol with the lightness and speed necessary for a distributed collaborative hypermedia information system.
- It is a generic stateless object-oriented protocol,
  - May be used for many similar tasks
    - E.g. name servers, and distributed object-oriented systems, by extending the commands, or "methods", used.

http://www.w3.org/Protocols/HTTP/HTTP2.html

HTTP Protocol (ctd)

- A feature of HTTP is the negotiation of data representation, allowing systems to be built independently of the development of new advanced representations.

Connections in HTTP

- The http protocol is designed for client server architectures
- The protocol is basically stateless, a transaction consists of:
  - Connection
    - The establishment of a connection by the client to the server
  - Request
    - The sending, by the client, of a request message to the server;
  - Response
    - The sending, by the server, of a response to the client;
  - Close
    - The closing of the connection by either both parties.

we have seen request and response as parameter of any callback to the server in NodeJS

http://www.w3.org/Protocols/HTTP/HTTP2.html

HTTP: GET and POST methods

- The GET method means "retrieve whatever information (…) is identified by the Request-URI".
  - e.g. Your browser requires a page (e.g. containing a form) from a server using a GET method
- The POST method is used to request that the origin server accepts the entity enclosed in the request [and acts upon it].
  - e.g. the browser POSTs the values for a form to the server

http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Request Headers

- When an HTTP client sends a request, it is required to supply a request line (usually GET or POST).
- It can also send a number of other headers, all of which are optional
  - Except for Content-Length required for POSTs.
- Here are the most common headers:
  - Accept: The MIME types the client (e.g. browser) prefer
  - Accept-Charset: The character set the client expects.
  - Accept-Encoding:
    - The types of data encodings (such as gzip) the client knows how to decode.

Common examples [edit]

- `application/json`
- `application/x-www-form-urlencodec`
- `multipart/form-data`
- `text/html`

http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/

Request Headers (2)

- Accept-Language
  - The language the client is expecting, in case the server has versions in more than one language.
- Authorization, Authorization info,
  - Usually in response to a WWW-Authenticate header from the server.
- Content-Length
  - Obligatory for POST messages, how much data is attached
- Cookie
- From
  - email address of requester; only used by Web spiders and other custom clients, not by browsers
- Host
  - Host and port as listed in the ***original*** URL

Request Headers (3)

- If-Modified-Since
  - Only return documents newer than this, otherwise send a 304 "Not Modified" response
- Referrer
  - The URL of the page containing the link the user followed to get to current page
- User-Agent
  - Type of client (robot, browser, etc.)
  - Useful if server is returning client-specific content
  - Useful to identify the client: particularly important for Spiders
- Others:
  - UA-Pixels, UA-Color, UA-OS, UA-CPU (nonstandard headers sent by some Internet Explorer versions, indicating screen size, color depth, operating system, and cpu type used by the client's system)

Please note!

- Most of these headers are used to reduce the server bandwidth consumption
  - The four Accept* headers
  - If-modified since

Please note very often you will use the http request directly (e.g. via a direct call). In other cases you will use some forms of Web API (e.g. Twitter API). However all these APIs have equivalent information that is either set up automatically (e.g. user agent in the Twitter API) or via the parameters of the API itself

Not setting these means low marks in the assignment!

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Responses Codes

http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/

Status Code	Associated Message	Meaning
100	Continue	Continue with partial request. (New in HTTP 1.1)
101	Switching Protocols	Server will comply with `Upgrade` header and change to different protocol. (New in HTTP 1.1)
200	OK	Everything's fine; document follows for `GET` and `POST` requests. This is the default for servlets; if you don't use `setStatus`, you'll get this.
201	Created	Server created a document; the `Location` header indicates its URL.
202	Accepted	Request is being acted upon, but processing is not completed.
203	Non-Authoritative Information	Document is being returned normally, but some of the response headers might be incorrect since a document copy is being used. (New in HTTP 1.1)
204	No Content	No new document; browser should continue to display previous document. This is a useful if the user periodically reloads a page and you can determine that the previous page is already up to date. However, this does not work for pages that are automatically reloaded via the Refresh response header or the equivalent `<META HTTP-EQUIV="Refresh" ...>` header, since returning this status code stops future reloading. JavaScript-based automatic reloading could still work in such a case, though.
205	Reset Content	No new document, but browser should reset document view. Used to force browser to clear CGI form fields. (New in HTTP 1.1)
206	Partial Content	Client sent a partial request with a `Range` header, and server has fulfilled it. (New in HTTP 1.1)
300	Multiple Choices	Document requested can be found several places; they'll be listed in the returned document. If server has a preferred choice, it should be listed in the `Location` response header.
301	Moved Permanently	Requested document is elsewhere, and the URL for it is given in the `Location` response header. Browsers should automatically follow the link to the new URL.

303 See Other: The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The new URI is not a substitute reference for the originally requested resource. The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.

http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/

Responses (2)

401	Unauthorized	Client tried to access password-protected page without proper authorization. Response should include a `WWW-Authenticate` header that the browser would use to pop up a username/password dialog box, which then comes back via the `Authorization` header.
403	Forbidden	Resource is not available, regardless of authorization. Often the result of bad file or directory permissions on the server.
404	Not Found	No resource could be found at that address. This is the standard "no such page" response. **This is such a common and useful response that there is a special method for it in `HttpServletResponse`: `sendError(message)`.** The advantage of `sendError` over `setStatus` is that, with `sendError`, the server automatically generates an error page showing the error message.
405	Method Not Allowed	The request method (`GET`, `POST`, `HEAD`, `DELETE`, `PUT`, `TRACE`, etc.) was not allowed for this particular resource. (New in HTTP 1.1)
406	Not Acceptable	Resource indicated generates a MIME type incompatible with that specified by the client via its `Accept` header. (New in HTTP 1.1)
407	Proxy Authentication Required	Similar to 401, but proxy server must return a `Proxy-Authenticate` header. (New in HTTP 1.1)
408	Request Timeout	The client took too long to send the request. (New in HTTP 1.1)
409	Conflict	Usually associated with `PUT` requests; used for situations such as trying to upload an incorrect version of a file. (New in HTTP 1.1)
410	Gone	Document is gone; no forwarding address known. Differs from 404 in that the document is is known to be permanently gone in this case, not just unavailable for unknown reasons as with 404. (New in HTTP 1.1)
411	Length Required	Server cannot process request unless client sends a `Content-Length` header. (New in HTTP 1.1)
412	Precondition Failed	Some precondition specified in the request headers was false. (New in HTTP 1.1)
413	Request Entity Too Large	The requested document is bigger than the server wants to handle now. If the server thinks it can handle it later, it should include a `Retry-After` header. (New in HTTP 1.1)

http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/

Res

414	Request URI Too Long	The URI is too long. (New in HTTP 1.1)
415	Unsupported Media Type	Request is in an unknown format. (New in HTTP 1.1)
416	Requested Range Not Satisfiable	Client included an unsatisfiable `Range` header in request. (New in HTTP 1.1)
417	Expectation Failed	Value in the `Expect` request header could not be met. (New in HTTP 1.1)
500	Internal Server Error	Generic "server is confused" message. It is often the result of CGI programs or (heaven forbid!) servlets that crash or return improperly formatted headers.
501	Not Implemented	Server doesn't support functionality to fulfill request. Used, for example, when client issues command like `PUT` that server doesn't support.
502	Bad Gateway	Used by servers that act as proxies or gateways; indicates that initial server got a bad response from the remote server.
503	Service Unavailable	Server cannot respond due to maintenance or overloading. For example, a servlet might return this header if some thread or database connection pool is currently full. Server can supply a `Retry-After` header.
504	Gateway Timeout	Used by servers that act as proxies or gateways; indicates that initial server didn't get a response from the remote server in time. (New in HTTP 1.1)
505	HTTP Version Not Supported	Server doesn't support version of HTTP indicated in request line. (New in HTTP 1.1)

Never ignore the response code of a request!!!

Why are error codes important?

https://www.wordbee.com/blog/localization-industry/10-mistranslated-signs-that-show-the-importance-of-translation/

Lost in translation: road sign carries email reply

https://www.theguardian.com/theguardian/2008/nov/01/5

▲ The Welsh language road sign reading: 'I am out of the office at the moment', erected in Swansea. Photograph: PA

A council put up a Welsh language road sign reading "I am out of the office at the moment" when it should have said "No entry for heavy goods vehicles".

Swansea council contacted its in-house translation service when designing the bilingual sign. The seeds of confusion were sown when officials received an automated email response in Welsh from an absent translator, saying: "I am not in the office at the moment. Please send any work to be translated."

Unaware of its real meaning, officials had it printed on the sign. The council

Https

- Hypertext transfer protocol secure (HTTPS) is the secure version of HTTP
  - HTTPS is encrypted in order to increase security of data transfer
  - It is particularly important when sensitive data is sent
    - bank account information, emails, etc.
- Any website, especially those that require login credentials, should use HTTPS
  - in the current web browsers such as Chrome, websites that do not use HTTPS are marked differently than those that are secure

Https

- HTTPS uses an encryption protocol to encrypt communications
  - Transport Layer Security (TLS) - Formerly known as Secure Sockets Layer (SSL)
- It secures communications by using an **asymmetric public key infrastructure**.
  - It uses two different keys to encrypt communications between two parties:
    - The private key
      - controlled by the owner - it is kept private. Used to decrypt information encrypted by the public key
    - The public key
      - available to everyone who wants to interact with the server securely
      - Information that's encrypted by the public key can only be decrypted by the private key

SSL Certificates

- SSL Certificates are small data files that digitally bind a cryptographic key to an organisation's identity
  - they bind together:
    - A domain name, server name or hostname.
    - An organisational identity (i.e. company name) and location
  - The certificate contain
    - the keys pair
    - the "subject," i.e. the identity of the certificate/website owner
    - the certificate is verified by an external authority
- You can get your free certificate from letsencrypt.org
  - a non profit authority

http://www.theguardian.com/technology/2015/feb/18/http2-speed-up-web-browsing-desktop-mobile

HTTP/2

http://en.wikipedia.org/wiki/HTTP/2

- HTTP/2 keeps most of HTTP 1.1's high level syntax,
  - Methods, status codes, header fields, and URIs.
- The element that is modified is
  - How data is framed and transported between the client and the server.
- Websites that are efficient minimise the number of requests required to render an entire page by minifying
  - reducing the amount of code and packing smaller pieces of code into bundles,
  - (without reducing its ability to function) resources such as images and scripts.

HTTP/2 ctd

http://en.wikipedia.org/wiki/HTTP/2

- However, minification is not necessarily convenient nor efficient,
  - it may still require separate HTTP connections to get the page and the minified resources.
- HTTP/2 allows the server to "push" content
  - to respond with data for more queries than the client requested.
  - It allows servers to supply data it knows web browser will need to render a web page, without waiting for the browser to examine the first response, and without the overhead of an additional request cycle
- Additional performance improvements come from
  - multiplexing of requests and responses to avoid the head-of-line blocking problem in HTTP 1
  - header compression, and prioritization of requests

Going beyond the limitations of http

- the protocol has been fantastic and brought the Web from a text only document linkage to a very sophisticated environment
  - However, its limitations have been a main issue over the past few year, especially:
    - memoryless
    - client-initiated
- Today we will see how we go beyond that

What you should remember

- How a connection is created with the http protocol
  - connection, request, response, closure
  - as this is relevant to most APIs you will ever use
- The main error codes
  - never ignore the error code
  - remember the guy who ignored an away message and printed it in Welsh

Questions?

Http: Streaming API

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

HTTP/1.1 Enhancements

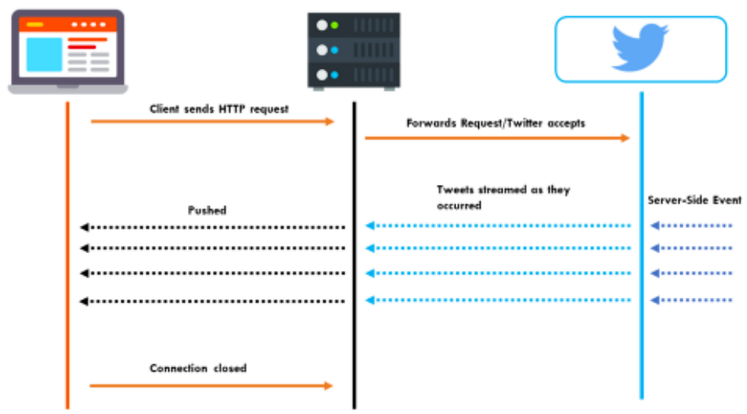
https://medium.com/platform-engineer/web-api-design-35df8167460

- It implementa critical performance optimisations and feature enhancements, e.g.
  - persistent and pipelined connections,
  - chunked transfers,
  - new header fields in request/response body etc.
- Two main headers
  - most of the modern improvements to HTTP rely on these two:
    - Keep-Alive
      - to set policies for long-lived communications between hosts (timeout period and maximum request count to handle per connection)
    - Upgrade
      - to switch the connection to an enhanced protocol mode such as HTTP/2.0 (h2,h2c) or Websockets (websocket)

Client-server communication

- RESTful APIs:
  - the most common APIs: the client requests a service and the server satisfies it
    - request/response model
- HTTP Polling
  - the client polls the server requesting new information
    - HTTP Short Polling: practically never used as it procures intense traffic
    - HTTP Long Polling: rather expensive but used
    - HTTP Periodic Polling: With a predefined time gap between two requests
    - HTTP Streaming: the most used, provides a long-lived connection for instant and continuous data push
      - The server trickles out a response of indefinite length (it's like polling infinitely).
      - HTTP streaming is performant, easy to consume and can be an alternative to WebSockets

HTTP Streaming example: Twitter



https://medium.com/platform-engineer/web-api-design-35df8167460

Streaming in Node/MySQL

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

var connection = mysql.createConnection(
    {
      host     : 'mysql_host',
      user     : 'your-username',
      password : 'your-password',
      database : 'database_name',
    }
);
connection.connect();
var query = connection.query('SELECT * FROM your_relation');

query.on('error', function(err) {
    throw err;
});

query.on('fields', function(fields) {
    console.log(fields);
});

query.on('result', function(row) {
    console.log('name: ' + row.name +
      ' ', row.surName);
});

query.on('end', function() {

// When it's done I Start something else
});
connection.end();
```

event received while processing: error

the list of fields in the next record

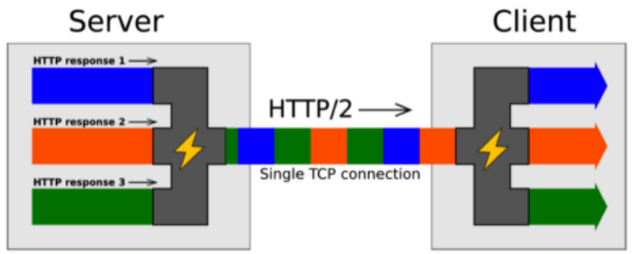
event received while processing: a row of data is available for processing
use elements from the fields variable to access parts of the row

when all rows have been received

HTTP/2 Server Push

https://medium.com/platform-engineer/web-api-design-35df8167460

- A mechanism for a server to proactively push assets (stylesheets, scripts, media) to the client cache in advance



Finally: WebSocket

We will discuss in the next section

Questions?

Bidirectional client-server architecture with socket.io

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

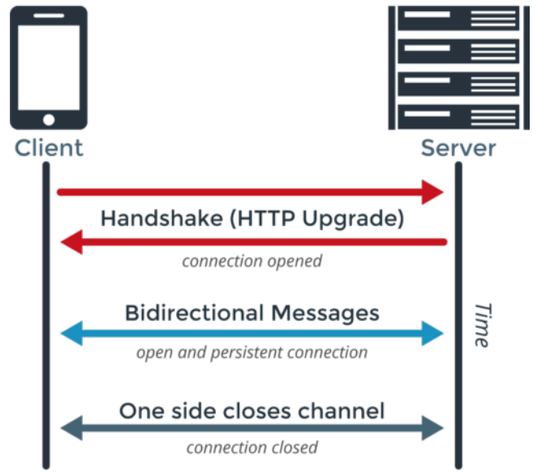
Sockets and Websockets

- A socket is a channel of communication between processes (on the same or different computers)
  - They create a persistent connection between the client and the server and both parties can start sending data at any time
- A Websocket establishes communication between two processes on different web connected machines via the TCP protocol
  - A web socket is defined by a URL and a port
    - a URL is an address, i.e. it represents a connected computer (like a street address, e.g. Fenton Road, Sheffield)
    - a port is an address number on that machine, (like a street number, e.g. 55 Fenton Road, Sheffield)

Operations

- Like in the standard http protocol you open and close the communication.
  - you also send and receive data
  - a web socket is event based
    - the process waits for an event
      - the receiving of a communication
    - the process can raise events at any time on the partner machine
      - by sending data via the socket

WebSocket Communication



https://medium.com/platform-engineer/web-api-design-35df8167460

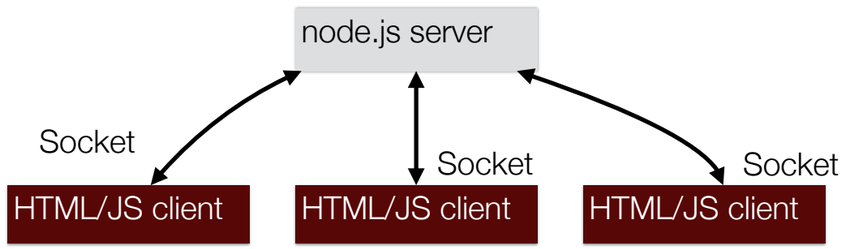
Socket.io

Socket.io

- Socket.IO enables real-time bidirectional event-based communication between browser and server
  - 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

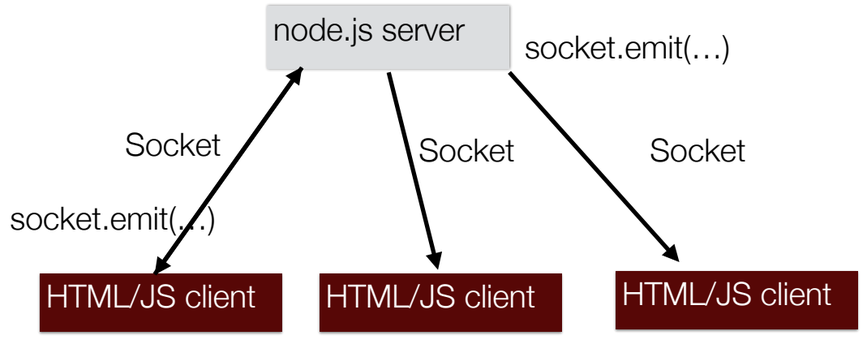
1 server, n clients, n sockets

- 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 in express

Express server routes/index.js

```
router.get('/', function(req, res, next) {
  res.render('index', { title: 'My Chat' });
```

HTML/JS client

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

client opens the socket and connects

```
socket.emit ('message', param)
```

client emits a message

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

client receives a message

server receives a message

server emits a message

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

callback

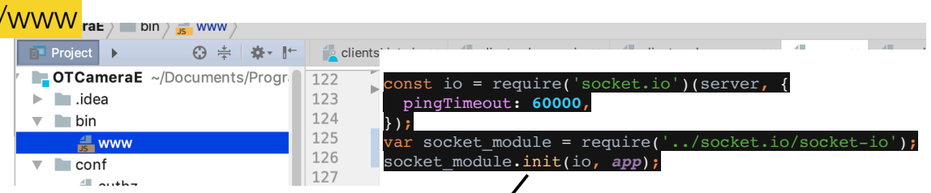
in /socket.io/socket.io.js

socket automatically closes when client navigates away from page

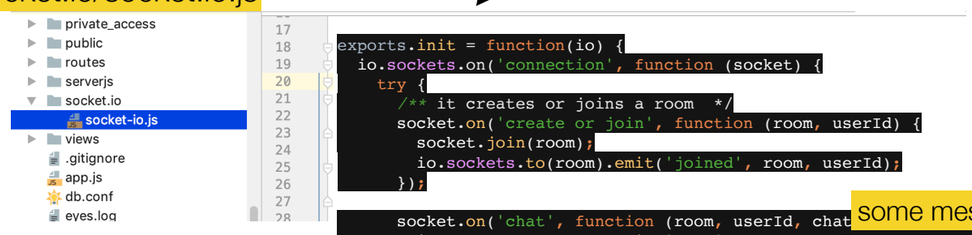
In WebStorm

- open the Terminal and run *npm install socket.io*

in /bin/www



in /socket.io/socket.io.js



some messages

- socket.io enables sending and receiving messages

```
exports.init = function(io) {
  io.sockets.on('connection', function (socket) {
    try {
      /**
       * it creates or joins a room
       */
      socket.on('create or join', function (room, userId) {
        socket.join(room);
        io.sockets.to(room).emit('joined', room, userId);
      });

      socket.on('chat', function (room, userId, chatText) {
        io.sockets.to(room).emit('chat', room, userId, chatText);
      });
```


In IntelliJ

in a Javascript script associated to the client

```
let socket = io();
...
/* it declares the expected messages and associated actions */
socket.on('joined', function (){
...
})
```


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');
  });
});
```

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

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>
```


Broadcasting

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

Broadcasting messages

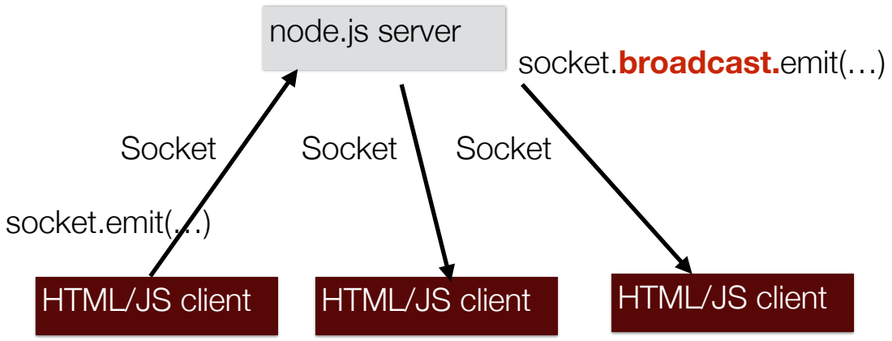
To broadcast, simply add a `broadcast` flag to `emit` and `send` method calls. Broadcasting means sending a message to everyone else except for the socket that starts it.

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

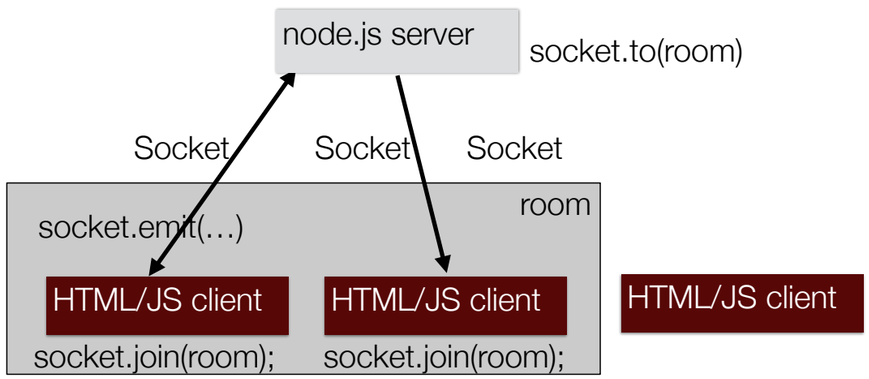
Each `Socket` in Socket.IO is identified by a random, unguessable, unique identifier `Socket#id`. For your convenience, each socket automatically joins a room identified by this id.

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



Connecting to room

- Client side:

```
function connectToRoom(roomNo, name) {
    socket.emit('create or join', roomNo, name);
}
```

- Server side (in /socket.io/socket-io.js)

```
socket.on('create or join', function (room, userId) {
  socket.join(room);
 });
```

Now the client is in the room

Namespaces

- Namespaces enable dedicated channels (e.g. like in Slack)

We have two namespaces here: /chat and /news

  - All users can access all channels

```
exports.init = function(io) {

  // the chat namespace
  const chat= io
      .of('/chat')
      .on('connection', function (socket) {
    try {
          socket.on('create or join', function (room, userId) {
...
```

in /socket.io/socket-io.js

```
// the news namespace
const news= io
      .of('/news')
      .on('connection', function (socket) {
    try {
          socket.on('create or join', function (room, userId) {
```


Namespaces

- On the client side we have the equivalent of multiple sockets

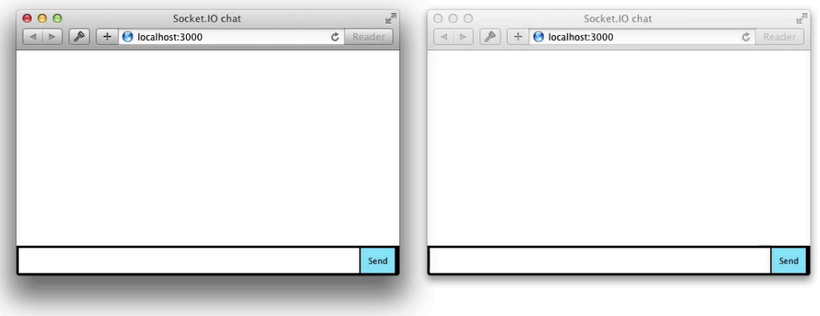
```
let chat= io.connect('/chat');
let news= io.connect('/news');
...
chat.on('joined', function (){
...
})

news.on('joined', function (){
...
})
```


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/



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

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 (or io.sockets.to)
broadcasts to all sockets in the room including the calling one

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 rest is just a form!

- We will see how to build a complete system in the lab

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!

What you should know

- how to create bidirectional client/server architectures using socket.io
- how to create the connection
- how to create a room and have different clients in different rooms
- the difference between sending messages and broadcasting
- how to build a simple chat system

Flexible Communication client server with Ajax

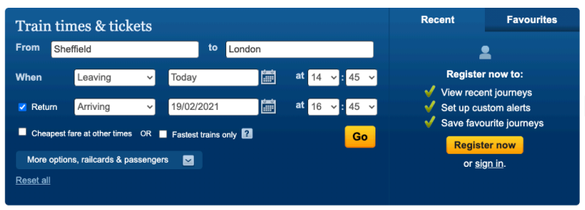
Prof. Fabio Ciravegna
Department of Computer Science
The University of Sheffield
f.ciravegna@shef.ac.uk

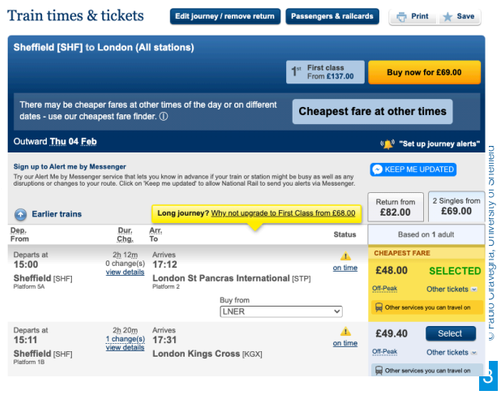
Client Server Communication

- In node JS we have seen the concept of event based computing
  - Long operations are executed in the background and raise an event when the operation is finished
    - Callbacks
    - Promises
- How about browsers?
  - For a long time we have event based communication via Javascript, e.g.
    - *document.getElementById("button1").onclick = function() {myFunction()};*
  - But how about communication with the client?

Classical communication

- generally based on the browser connecting to the server (e.g via a form) and waiting for an answer, e.g. a new page
  - While waiting for the response, the client is blocked





Towards a different Client-Server Architecture

Ajax: A New Approach to Web Applications
by Jesse James Garrett
http://adaptivepath.com/publications/essays/archives/000385.php

- Classic web application model
  - Most user actions in the interface trigger an HTTP request back to a web server.
  - The server does some processing — retrieving data, crunching numbers, talking to various legacy systems — and then returns an HTML page to the client.
- This approach makes a lot of technical sense, but it doesn't make for a great user experience.
  - While the server is doing its thing, what's the user doing?
    - That's right, waiting.
  - And at every step in a task, the user waits some more.
- Obviously, if we were designing the Web from scratch for applications, we wouldn't make users wait around. Once an interface is loaded, why should the user interaction come to a halt every time the application needs something from the server? In fact, why should the user see the application go to the server at all?

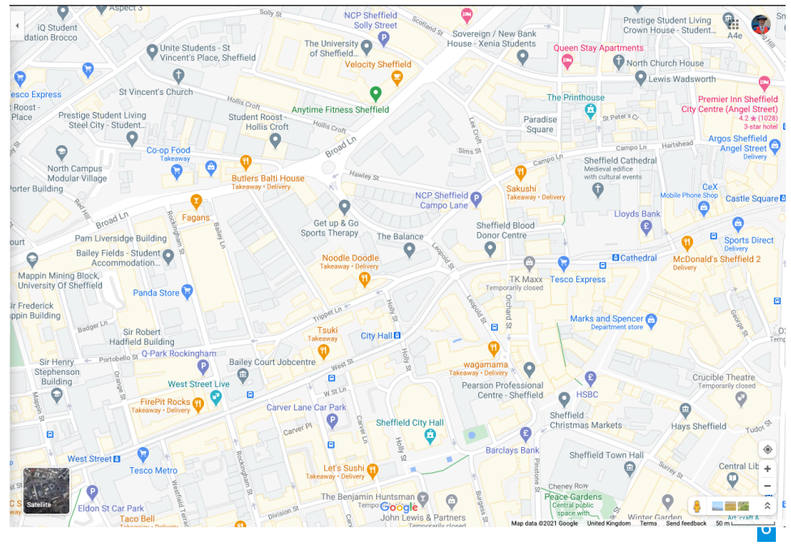
HTTP based = stateless+directional

- The communication implies sending a request (GET/POST) and receiving a full document to Display (e.g. HTML+CSS)
- Communication is stateless and directional (REST)
  - the next request will have to contain all the information because the server will have forgotten everything about my request
  - The client requests the information and the server responds
- For security reasons it is not possible to establish a communication where client and servers collaborate to achieve a task
  - e.g. the client asks for some data
    - and then asks for some more but without reissuing the request for the entire data
    - We have seen socket.io but in the traditional model there was no space for bidirectionally

Example: Mapping

- Suppose I have this map. Regent Court is not visible
  - it is just outside

it is more or less here

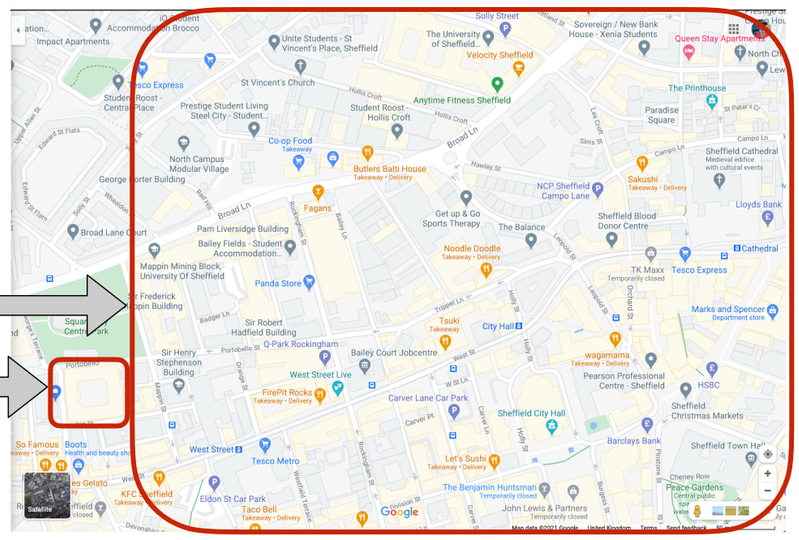


In a traditional rest interaction

- I would have to reissue the request for the entire map
  - 90% of the map I already had
  - it is a serious waste

Part I already had

Regent Court is here



The communication we would like

- We would like the client
  - to be able to request just the missing tiles of the map and reuse those that are already there
  - to allow to use the map while the missing tiles are fetched
- Advantages:
  - Less traffic
  - The browser page would not freeze while the data is fetched
    - Better user experience
- We can do that with Ajax
  - a Javascript middleware built in in the browser which intercepts the requests to the server and may help optimising them
    - being event based, while communication is ongoing, the browser is still alive and usable by the user

AJAX

- Ajax is a Javascript library built-in in the browser
- It provides a filter between any html/javascript request and any server answer

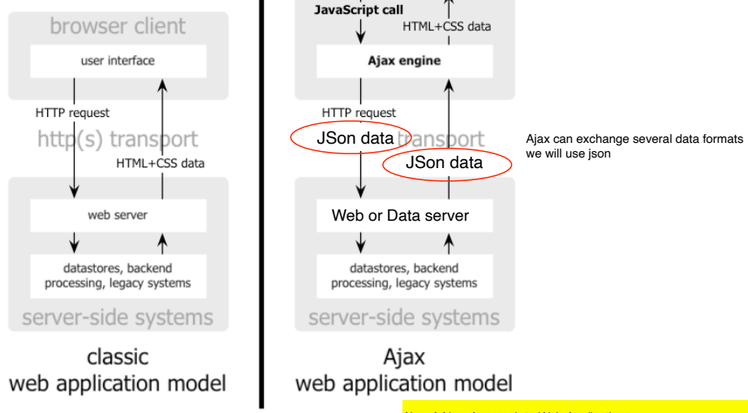
- It allows
  - exchanging data between browser and server
    - rather than just fetching routes with serialised data and returning html
  - detaching the browser when a query is sent to the server
    - so that the user can still interacts while the data is fetched

Ajax is different

Ajax: A New Approach to Web Applications
by Jesse James Garrett
http://adaptivepath.com/publications/essays/archives/000385.php

- An Ajax application eliminates the start-stop-start-stop nature of interaction on the Web by introducing an intermediary — an Ajax engine — between the user and the server
- The Ajax engine is built in in the browser it will
  - render the user interface
  - communicate with the server on the user's behalf.
- The Ajax engine allows the user's interaction with the application to happen asynchronously
  - independent of communication with the server.
- The user is never staring at a blank browser window and an hourglass icon, waiting around for the server to do something

Ajax exchanges data rather than HTML pages



Ajax: A New Approach to Web Applications
by Jesse James Garrett
http://adaptivepath.com/publications/essays/archives/000385.php

What does the Ajax engine do?

- Every user action that normally would generate an HTTP request takes the form of a JavaScript call to the Ajax engine instead.
- Any response to a user action that doesn't require a trip back to the server — such as simple data validation, editing data in memory, and even some navigation — the engine handles on its own.
- If the engine needs something from the server in order to respond
  - if it's submitting data for processing, loading additional interface code, or retrieving new data
  - the engine makes those requests asynchronously, usually using JSon, without stalling a user's interaction with the application.
- Example:
  - visualise a Google Map http://maps.google.com/
  - grab the top right corner of the map
  - pull towards left bottom corner
  - on a slow connection, you will see the missing tiles as blank but the rest of the page works normally

The original Ajax code

We will not use this

Create object

Post to server

What to do when server responds
(processChange is a function that will be called)

```
<html><head>
    <script>
    function submitForm() {
       var xhr;
       try {  xhr = new ActiveXObject('Msxml2.XMLHTTP');   }
       catch (e)     {
          try {   xhr = new ActiveXObject('Microsoft.XMLHTTP');    }
          catch (e2)        {
            try {  xhr = new XMLHttpRequest();     }
            catch (e3) {  xhr = false;   }
          }    }
       xhr.onreadystatechange  = processChange;
       xhr.open(POST, "data.txt",  true);
       xhr.setRequestHeader("Content-type","application/x-www-form-urlencoded");
       var namevalue=encodeURIComponent(document.getElementById("name").value)
       var agevalue=encodeURIComponent(document.getElementById("age").value)
       var parameters="name="+namevalue+"&age="+agevalue
        xhr.send(parameters)

    }

function processChange()    {
       if(xhr.readyState  == 4)       {
          if(xhr.status  == 200)
             document.ajax.dyn="Received:"  + xhr.responseText;
          else
            document.ajax.dyn="Error code " + xhr.status;
    }    };

    </script>
    </head>
    </body> </html>
```

When the function terminates, the request has been sent and the browser is free to do something else

When server responds, do something

We will use JQuery

What is jQuery?

jQuery is a lightweight, "write less, do more", JavaScript library.

The purpose of jQuery is to make it much easier to use JavaScript on your website.

jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code.

jQuery also simplifies a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

The jQuery library contains the following features:

- HTML/DOM manipulation
- CSS manipulation
- HTML event methods
- Effects and animations
- AJAX
- Utilities

http://www.w3schools.com/jquery/jquery_intro.asp

Tip: In addition, jQuery has plugins for almost any task out there.

If you do not know JQuery, see the Web Technology class slides

The Generic operation

```
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"><
</head>
…

<script>
sendAjaxQuery(url, data){
    $.ajax({
        url: url,
        type: "POST",
        data: data,

        contentType: 'application/json',
        error: function (...) {
       // do something here
        },
        success: function (response) {
       // do something here
        }
    });}
</script>
```

the url to contact

declare the action (POST)

the data to send (no need to stringily if declared datatype is JSON - done automatically by jQuery)

declare a Json interaction

if an error is returned (http response code >= 300)

http response code 200<=x<300

Ajax Operations: load

- The load() method loads data from a server and puts the returned data into the selected element.
  - *$(selector).load(URL,data,callback);*
  - e.g. *$("#div1").load("demo_test.txt");*
  - The optional callback parameter specifies a callback function to run when the load() method is completed. The callback function can have different parameters:
    - responseTxt - contains the resulting content if the call succeeds
    - statusTxt - contains the status of the call
    - xhr - contains the XMLHttpRequest object

https://www.w3schools.com/jquery/jquery_ajax_load.asp

GET vs. POST

- Get

$.get(URL,callback);

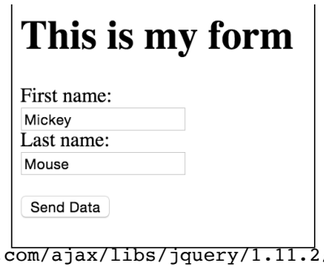
```
$("button").click(function(){
  $.get("demo_test.asp", function(data, status){
    alert("Data: " + data + "\nStatus: " + status);
  });
});
```

- Post will have also the post parameters

$.post(URL,data,callback);

```
$("button").click(function(){
  $.post("demo_test_post.asp",
  {
    name: "Donald Duck",
    city: "Duckburg"
  },
  function(data, status){
    alert("Data: " + data + "\nStatus: " + status);
  });
});
```


FORM




```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Ajax form</title>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
    </script>
</head>
<body>
<h1>This is my form</h1>
```

FORM MUST return false when Ajax is used

```
<form id="myForm" onsubmit="return false;" >
    First name:<br>
    <input type="text" name="firstname" value="Mickey">
    <br>
    Last name:<br>
    <input type="text" name="lastname" value="Mouse">
    <br><br>
    <button id="sendButton" onclick=sendData()>Send Data</button>
</form>
```

rather than using submit for form, just declare a button and the associated onclick event

Serialise form

```
function sendData() {
    var form = document.getElementById('myForm');
    sendAjaxQuery('/index.html', JSON.stringify(serialiseForm()));
}
```

Important: stringify

```
function serialiseForm(){
 var formArray= $("form").serializeArray();
 var data={};
 for (index in formArray){
    data[formArray[index].name]= formArray[index].value;
 }
 return data;
}
```

Form serialisation creates a structure containing all the data in the form.
The structure's fields names are the names of the input fields in the form

Ajax

data must be JSON stringify-ed before calling ajax

```
<script>
    function sendAjaxQuery(url, stringified-data) {
      $.ajax({
        url: url ,
        data: stringified-data,
        contentType: 'application/json',
```

You MUST declare type JSON

```
// Always use contentType: 'application/json' rather than dataType: 'json',
// as the latter may have unpredictable behaviour in my experience

        type: 'POST',
        success: function (dataR) {

  // no need to JSON parse the result, as we are using
    // contentType: 'application/json', so JQuery knows
    // it and unpacks the object before returning it

            var ret = dataR;
            // in order to have the object printed by alert
            // we need to JSON stringify the object
           // otherwise the alert will just print '[Object]'
            alert('Success: ' + JSON.stringify(ret));
        },
        error: function (xhr, status, error) {
            alert('Error: ' + error.message);
        }
    });
```

Server (routes/index.js)

```
var express = require('express');
var router = express.Router();
var bodyParser= require("body-parser");


/* GET home page. */
router.get('/postFile.html', function(req, res, next) {
  res.render('index', { title: 'My Form' });
});

router.post('/index.html', function(req, res, next) {
  var body= req.body; if we wanted to access the form fields, e.g.: req.body.firstname
    res.writeHead(200, { "Content-Type": "application/json"});
    res.end(JSON.stringify(body));
});


module.exports = router;
```

You MUST declare type JSON and Stringify before returning

Important!

- if while debugging you discover on either side (client or server) this situation:
  - you expect to receive an object like:

```
{"firstname":"Mickey","lastname":"Mouse","year":"1900","age":118}
```

  - and you receive in body-parser or by Ajax:

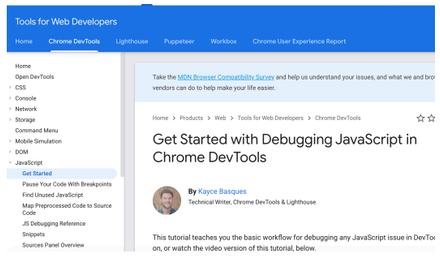
A value that makes no sense

A structure A field name that in reality is the content of your data structure in quotes

{ '{"firstname":"Mickey","lastname":"Mouse","year":"1900","age":118}' : ""}

Then it means that you are either **not stringify-ing the data** OR **not using contentType: 'application/json'** OR in general made a mess with **JSON stringify/parse**

https://developers.google.com/web/tools/chrome-devtools/javascript



Because of course you use a debugger when programming

If you do not, you have better learn everything about debugging professionally or becoming a hairdresser (which personally I feel it is the most pointless job in the world)

Whoever is caught debugging using printouts on the console only will be executed n the lab's podium ;)

Seriously: you must learn to debug as a pro - this will shorten your development time and will make your life easier

What you should know

- Why we need an event driven, flexible client server interaction with callbacks also in the communication between browser and server
- How to use Ajax (via JQuery)
- How to catch error codes and success codes
- How to create a callback to act on the results

Questions?

Axios: a library for Client-Server Communication

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

© Fabio Ciravegna, University of Sheffield

Axios

https://axios-http.com/docs/intro

- Axios is a promise-based HTTP Client for node.js and the browser
  - It is isomorphic (= it can run in the browser and nodejs with the same codebase).
    - On the server-side it uses the native node.js http module
    - on the client (browser) it uses XMLHttpRequests (i.e. Ajax)
- Make XMLHttpRequests from the browser
- Make http requests from node.js
- Supports the Promise API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data
- Client side support for protecting against XSRF
  - Cross-site request forgery, also known as one-click attack or session riding and abbreviated as CSRF (sometimes pronounced sea-surf[1]) or XSRF, is a type of malicious exploit of a website where unauthorized commands are submitted from a user that the web application trusts https://en.wikipedia.org/wiki/Cross-site_request_forgery

```
to add Axios to the nodes server:
        npm install axios
(or add the line:
    "axios": "^0.26.0",
to your package.json file
(see today's lab class)
```

To use Axios on the client (browser) side: add
`<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>`
to your html/EJS file!!

© Fabio Ciravegna, University of Sheffield

Performing a Get

```
axios.get('/user', {
    params: {
      ID: 12345
    }
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  })
  .then(function () {
    // always executed
  });
```

These params will be received in node.js's body

Code to execute in case of success

Code to execute in case of error

Code to execute in both cases at the end

© Fabio Ciravegna, University of Sheffield

Performing a POST

```
axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

These will be received in node.js's body (no need of the term "params")

Code to execute in case of success

Code to execute in case of error

Multiple parallel requests -> Promise.all

```
function getUserAccount() {
  return axios.get('/user/12345');
}

function getUserPermissions() {
  return axios.get('/user/12345/permissions');
}

Promise.all([getUserAccount(), getUserPermissions()])
  .then(function (results) {
    const acct = results[0];
    const perm = results[1];
  });
```

Define multiple functions returning an axios operaration

Promise.all them (use an array)

This will return a sorted array of results

A catch branch should be defined to cope with errors

Response: OK

```
{
  // `data` is the response that was provided by the server
  data: {},

  // `status` is the HTTP status code from the server response
  status: 200,

  // `statusText` is the HTTP status message from the server response
  // As of HTTP/2 status text is blank or unsupported.
  // (HTTP/2 RFC: https://www.rfc-editor.org/rfc/rfc7540#section-8.1.2.4)
  statusText: 'OK',

  // `headers` the HTTP headers that the server responded with
  // All header names are lower cased and can be accessed using the bracket notation.
  // Example: `response.headers['content-type']`
  headers: {},

  // `config` is the config that was provided to `axios` for the request
  config: {},

  // `request` is the request that generated this response
  // It is the last ClientRequest instance in node.js (in redirects)
  // and an XMLHttpRequest instance in the browser
  request: {}
}
```

The data

The code (always check in case of error!!)

When using then, you will receive the response as follows:

```
axios.get('/user/12345')
  .then(function (response) {
    console.log(response.data);
    console.log(response.status);
    console.log(response.statusText);
    console.log(response.headers);
    console.log(response.config);
  });
```


Response: Error

```
axios.get('/user/12345')
  .catch(function (error) {
    if (error.response) {
      // The request was made and the server responded with a status code
      // that falls out of the range of 2xx
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else if (error.request) {
      // The request was made but no response was received
      // `error.request` is an instance of XMLHttpRequest in the browser and an instance of
      // http.ClientRequest in node.js
      console.log(error.request);
    } else {
      // Something happened in setting up the request that triggered an Error
      console.log('Error', error.message);
    }
    console.log(error.config);
  });
```

The .then branch is not shown here

An example: the client

```
function sendAjaxQuery(url, data) {
    axios.post(url , data)
        .then (function (dataR) {
            // do something with the data
        })
        .catch( function (response) {
            alert (response.toJSON());
        })
}
```

the data is a javascript object that will be stringified automatically by axios

An example: NodeJS

never use the parameters without checking that they are valid (I have not done it here)

```
.post(function  (req, res) {
        axios.post('http://localhost:3000/add',
             {name: req.body.name, surname: req.body.surname})
             .then(received =>
                 res.json(received.data))
             .catch(err => {
                 res.setHeader('Content-Type', 'application/json');
                 res.status(505).json(err)
             })
});
```

the result is in the data field

Always check for errors

What you should know

- You should be able to perform an Ajax request using Axios
- You can use Axios to replace the fetch library we used in the last lab
  - much easier!!
- In the lab
  - You will be asked to perform the same exercise with JQuery's Ajax and with Axis

Video Communication via WebRTC

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

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

ABOUT TOPICS OF INTEREST HOW WE WORK INTERNET STANDARDS

> IETF News

Filter by topic and date

Topic: All | Date from: dd/mm/yyyy | Date to: dd/mm/yyyy | Filter

WebRTC technologies prove to be essential during pandemic

Grant Gross | IETF Blog Reporter

8 Dec 2020

WebRTC may arguably be the most important set of technologies used during the COVID-19 pandemic. All web-based videoconferencing services make use of WebRTC, a large set of technologies that allow web browsers to make voice, video, and real-time data calls. WebRTC protocols were developed at the IETF

The Pandemic

- The pandemic has made videoconferencing an essential requirement
  - in a few months it has revolutionised (among the many)
    - working patterns
    - health and wellbeing
    - school and education
- The world has effectively continued working thanks to the availability of WebRTC

W3C IETF FOR IMMEDIATE RELEASE

Web Real-Time Communications (WebRTC) transforms the communications landscape; becomes a World Wide Web Consortium (W3C) Recommendation and multiple Internet Engineering Task Force (IETF) standards

WebRTC enables rich, interactive, live voice and video communications anywhere on the Web, boosting global interconnection

Read testimonials from W3C Members

Translations I W3C Press Release Archive

https://www.w3.org/ and https://www.ietf.org/ — 26 January 2021 — The World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) announced today that Web Real-Time Communications (WebRTC), which powers myriad services, is now an official standard, bringing audio and video communications anywhere on the Web.

WebRTC, comprised of a JavaScript API for Web Real-Time Communications and a suite of communications protocols, allows any connected device, on any network, to be a potential communication end-point, on the Web. WebRTC already serves as a cornerstone of online communication and collaboration services.

WebRTC

https://www.w3.org/2021/01/pressrelease-webrtc-rec.html.en

Everyone uses WebRTC

Update September 2019: WebRTC DataChannel

As Mozilla's Nils Ohlmeier pointed out, Zoom switched to using WebRTC DataChannels for transferring media:

Nils Ohlmeier
@nilsohlmeier

Looks like @zoom_us has switched it's web client from web sockets to #WebRTC data channels. Performance a lot better compared to their old web client.

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 was supported by the W3C, Google, Mozilla, and Opera
  - supported by the major browsers

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

https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

- navigator.mediaDevices.getUserMedia
  - Webcam and microphone input are accessed without a plugin
    - (not to be confused with the old navigator.getUserMedia which is deprecated)

```
navigator.mediaDevices.getUserMedia(constraints)
.then(function(stream) {
  /* use the stream */
})
.catch(function(err) {
  /* handle the error */
});
```


getUserMedia contraints

constraints
A `MediaStreamConstraints` object specifying the types of media to request, along with any requirements for each type.

The *`constraints`* parameter is a `MediaStreamConstraints` object with two members: `video` and `audio`, describing the media types requested. Either or both must be specified. If the browser cannot find all media tracks with the specified types that meet the constraints given, then the returned promise is rejected with `NotFoundError`.

The following requests both audio and video without any specific requirements:

```
{ audio: true, video: true }
```

If `true` is specified for a media type, the resulting stream is *required* to have that type of track in it.

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 and that is mandatory
    - otherwise the system may decide to degrade the quality

```
{
  audio: true,
  video: { width: 1280, height: 720 }
}
```

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


- You can definitely do something very sophisticated

HTML5 container for vide

```
<video autoplay></video>
<script>
function prepareVideo(camid) {
    var session = {
        audio: true,
        video: {
            // if camera id not null, use it, otherwise select any
            // deviceId: camid ? {exact: camid} : true,
            // to choose the back camera use: (it only works on phones)
            facingMode: 'environment',
            // needs a minimum frame rate or it will not work
            //https://github.com/webrtc/samples/issues/922
            frameRate: {min: 10 },},};

    navigator.mediaDevices.getUserMedia(session)
        .then(async mediaStream => {
                // Chrome crbug.com/711524 requires await sleep
                await sleep(1000);
                gotStream(mediaStream);})
        .catch(function (e) {
            alert('Not supported on this device. Update your browser: ' + e.name);
     });}
</script>
```

callback function returns a videostream

remember never declare Javascript code within an HTML file. I do it for compact reading. Always link a Javascript file!

Choosing the camera

When using a phone and want to *ask for* the front camera, use:

- every constraint to getUserMedia is a request under the if-possible rule. So for example in computers this will default to the only camera they have

```
{ audio: true, video: { facingMode: "user" } }
```

To ask for the rear camera, use:

```
{ audio: true, video: { facingMode: "environment" } }
```

There is a possibility to use the keyword require: however it is not suggested as it would cause problems if the condition is not satisfied (crash or unavailability of video)

Choosing any camera or audio

```
cameraNames=[]; cameras = [];
function getSources(sourceInfos) {
    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.push(text);
            cameras.push(sourceInfo.id);
       } else if (sourceInfo.kind === 'audio') {
          audioSource = sourceInfo.id;
        }    }
    videoSource = cameras[cameras.length - 1];
}


MediaStreamTrack.getSources(getSources);
```

choosing (2)

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


Displaying the stream

In your HTML file declare a video element with autoplay:

```
<video id='video' autoplay></video>
```

In your Javascript file:

```
navigator.mediaDevices
    .getUserMedia({ audio: true, video: true })
    .then(gotStream(stream));

...

function gotStream(stream) {
    // enumerate the devices now as in iOS they will not have been available until now
    navigator.mediaDevices.enumerateDevices().then(initAudioVideo);

    //assigning the shared screen to the video element
    let mediaElement = document.getElementById('video');
    mediaElement.srcObject = stream;
```


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);
        }
    });
```

it creates a base64 image and stores it in an HTML5 blob

Server side

or you can save the base64 image directly in mongoDB. Not suggested uses a lot of space!

```
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: ''}));
        });    });
```


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>
```


Connecting to a peer

- So far we have just captured and displayed the local video and audio streams
  - We now have to connect to a party and start sending and receiving media
- This is made possible by socket.io

RTCPeerConnection

- The **`RTCPeerConnection`** interface represents a WebRTC connection
  - between the local computer
  - and a remote peer
- It provides methods to
  - connect to a remote peer,
  - maintain and monitor the connection,
  - close the connection once it's no longer needed

The `RTCPeerConnection()` constructor returns a newly-created `RTCPeerConnection`, which represents a connection between the local device and a remote peer.https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection


```
let pc = new RTCPeerConnection();
pc.ontrack = function(stream){
    let mediaElement = document.getElementById('remoteVideo');
    mediaElement.removeAttribute('autoplay');
    mediaElement.srcObject = stream[0];
};
pc.onremovestream = handleRemoteStreamRemoved;

pc.createOffer()
        .then(sendMessage)
        .catch(handleCreateOfferError);
}

/**
here we use socket.io to send the offer
*/
function sendMessage(connectionDescription) {
   pc.setLocalDescription(connectionDescription)
        .then(function () {
            // using socket.io to send the description
             socket.emit('message',
                // we want to add some credentials to avoid anybody to connect
                 getCredentials(),
                // we send the description of the connection
                 connectionDescription);
        })
        .catch(function(reason) {
            console.log("ERROR: in setLocalAndSendMessage "+reason);
        }); }
```

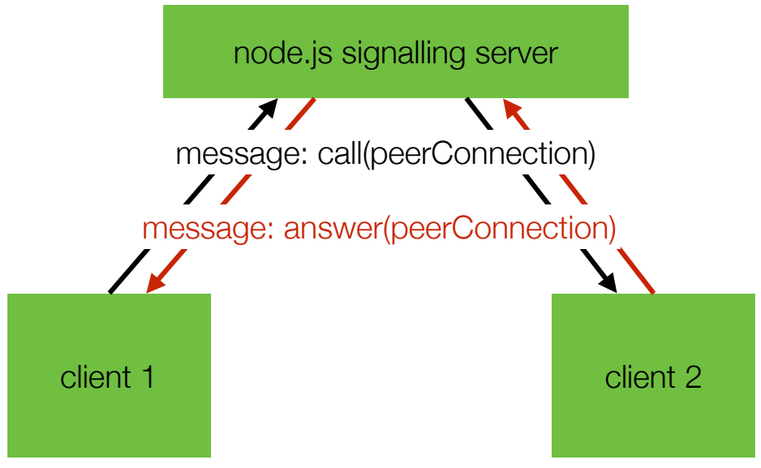
.ontrack is called when a stream is received from the peer via socket.io

called when the peer hangs up

it creates the offer for the peer and then sends it to the peer

it sets the local description to the RTCPeerConnection and then sends it to the peer via socket.io

Calling via socket.io



Calling

on the server, send it to the partner…

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

on the remote client…

```
// This client receives a message
socket.on('message', function(message) {
  // the client will receive an offer: it will store the remote description and will create
  // a description that will be sent as an answer
  if (message.type === 'offer') {
    createPeerConnection();
    // Line below - addStream is a method that has been removed from the standard and Safari
    // doesn't implement it - using addTrack instead
    // peerConnection.RTCPeerConnection.addStream(localStream);
    localStream.getTracks().forEach(track => peerConnection.addTrack(track, localStream));
    isStarted = true;
    peerConnection.setRemoteDescription(new RTCSessionDescription(message));
    createAnswer();
  }
  else if (message.type === 'answer' && peerConnection.isOfferer) {
    peerConnection.setRemoteDescription(new RTCSessionDescription(message));
  }
  else if (message.type === 'candidate' && peerConnection.isStarted) {
    addIceCandidate(message.candidate);  }
```

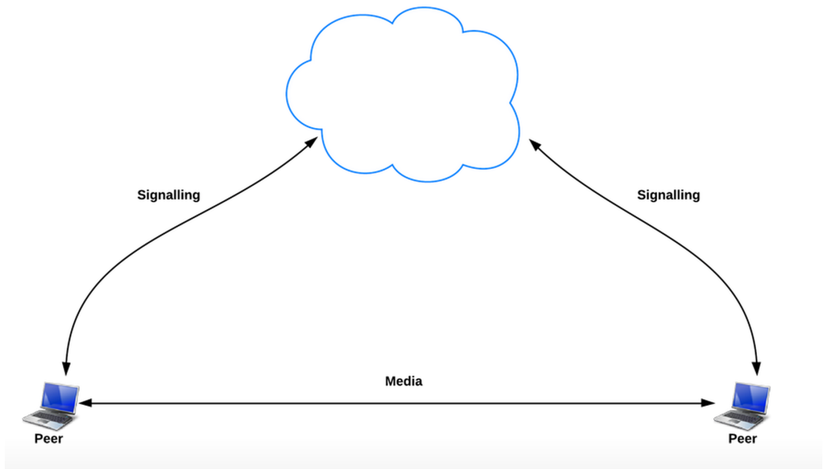
Firewalls!

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

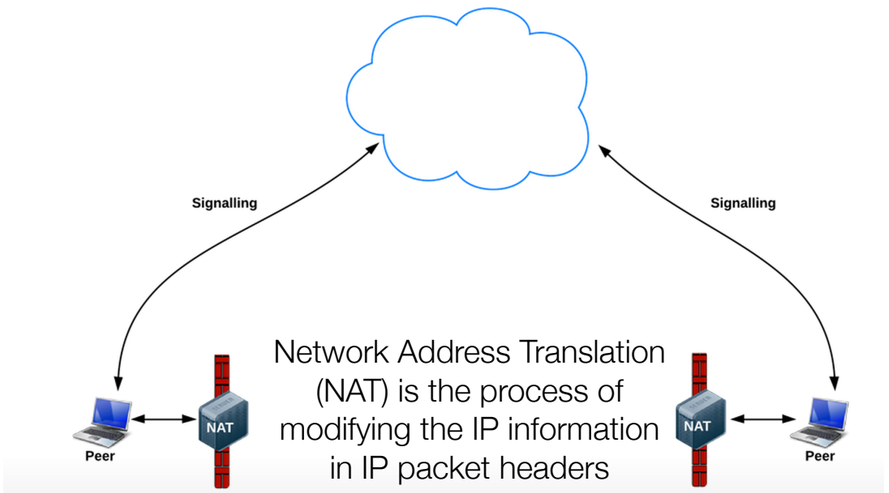
- A realistic situation however prevents this from working because firewalls prevent clients to see each other directly
- WebRTC needs four types of server-side functionality:
  - User discovery and communication.
  - Signalling.
  - NAT/firewall traversal.
  - Relay servers in case peer-to-peer communication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

An Ideal World

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



The Real World

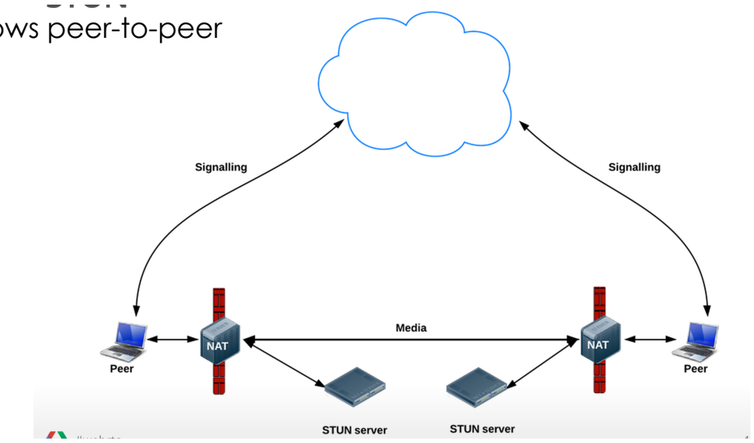


Network Address Translation (NAT) is the process of modifying the IP information in IP packet headers

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

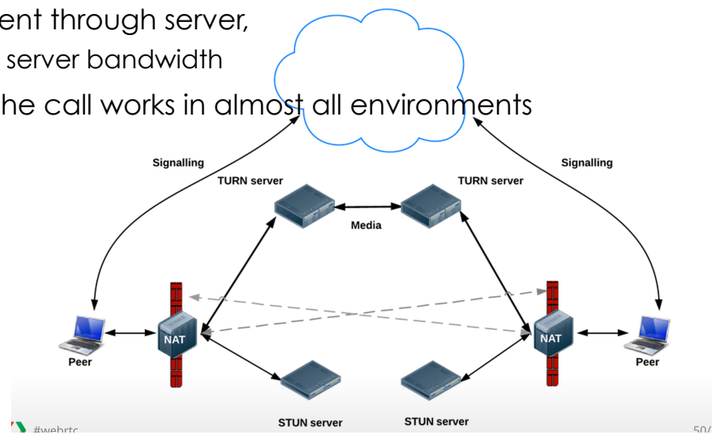
STUN Server

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



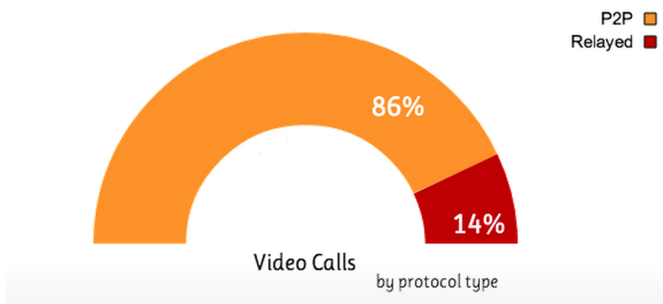
TURN Server

- 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



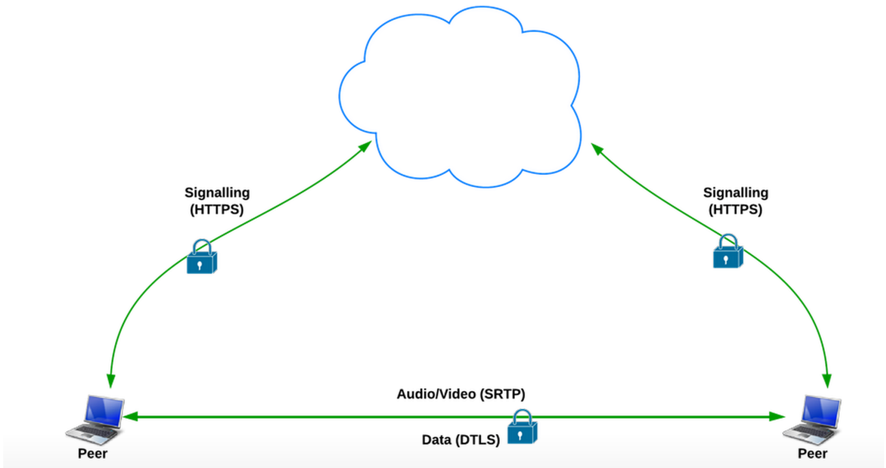
ICE

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



Always Use HTTPS

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



Adding a STUN server

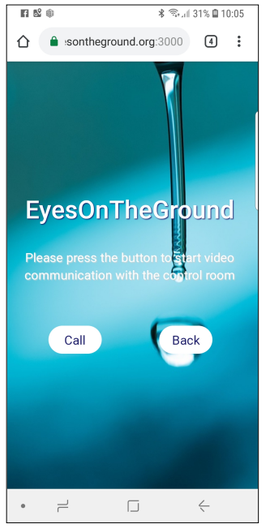
- We will not see the code of using ICE candidates
  - But in summary:

```
let pcConfig = { // stun server only
    'iceServers': [{"urls": "stun:stun.l.google.com:19302"}]
};

let pc= new RTCPeerConnection(pcConfig)
pc.onicecandidate= handleIceCandidate;


handleIceCandidate(event) {
    console.log('in icecandidate event: ', event);
    if (event.candidate) {
        SocketConnection.sendMessage(socket,
            {type: 'candidate', candidate: event.candidate, request);
    } else {
        console.log('End of candidates.');
    }
}
```

Remote Home Visits

ViVa

How it works



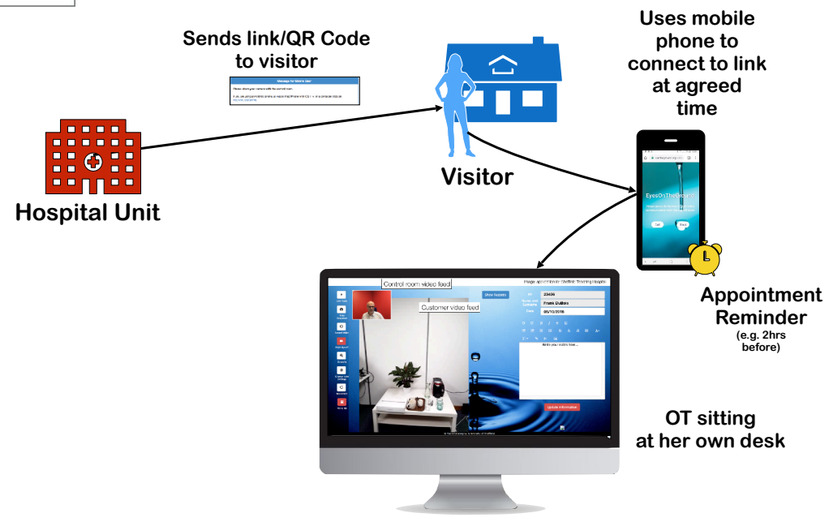
ViVa

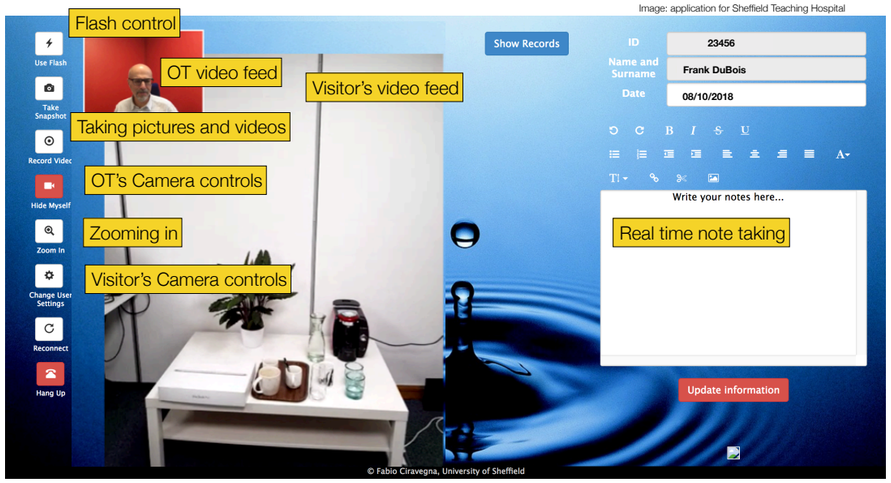
ViVa enables Remote Home Visits

Key to Occupational Therapy practice is understanding the person, the occupation and the environment

And how these three components interact together

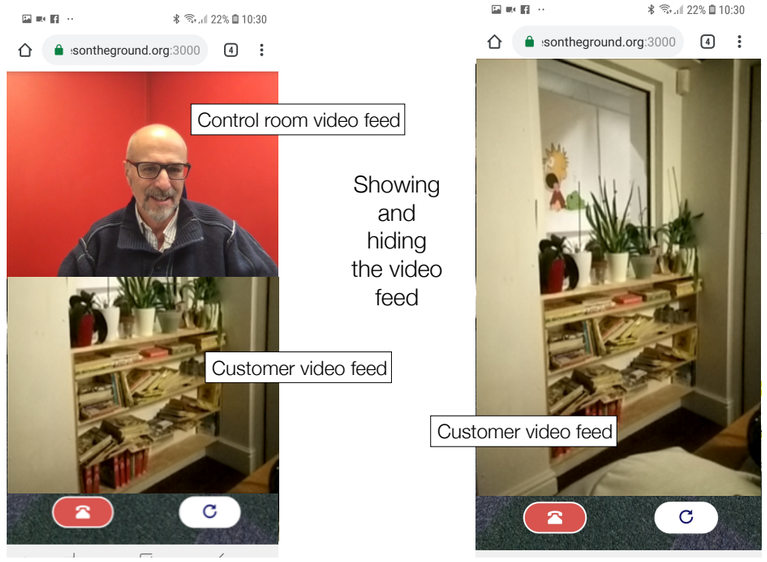
Home visits enable patients and their family an opportunity to assess, discuss and make plans to returning home

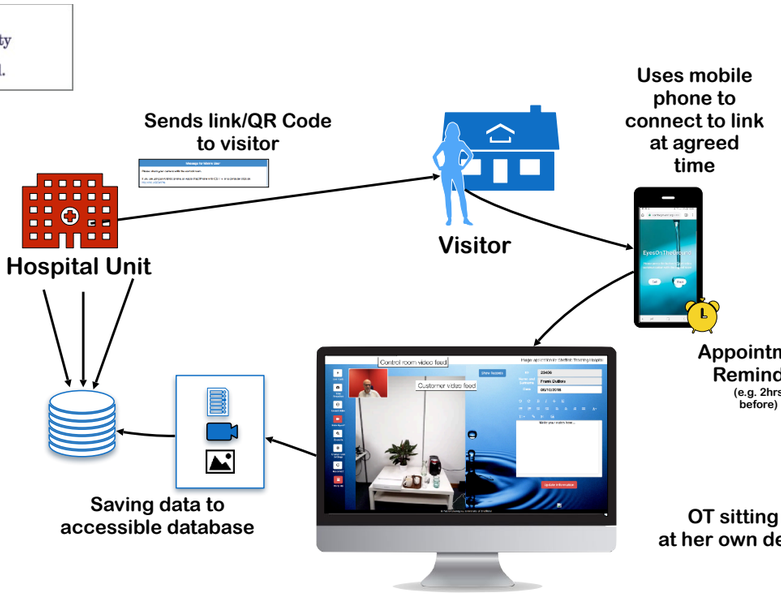


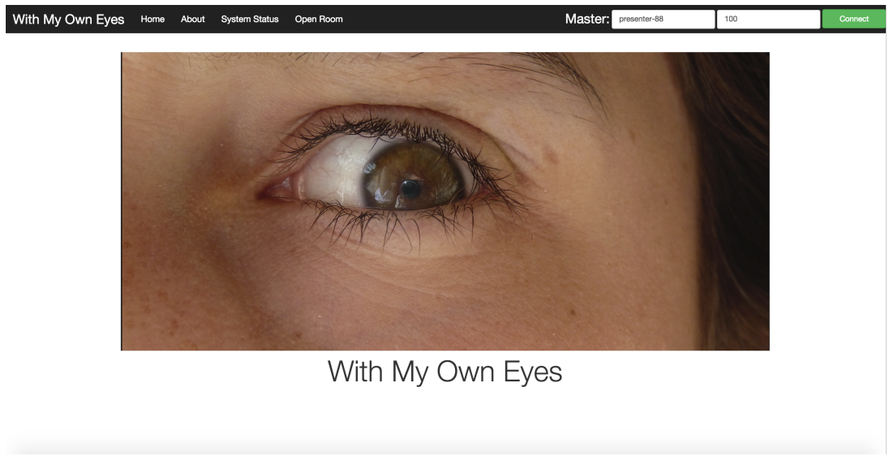


OT can see the video, record it, take photos, take notes live, etc.

The phone app during communication







With My Own Eyes

Exploitation (2): With My Own Eyes

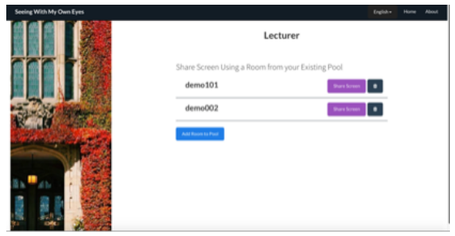
- A tool to support full participation of visually impaired students in lectures
- A large amount of information in student courses is delivered visually
  - A visually impaired student may therefore be at a significant disadvantage.
  - Staff are recommended to provide reading material and slides in 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

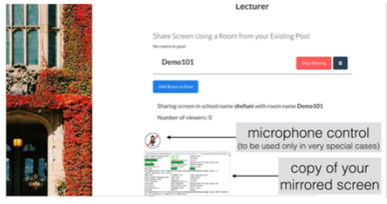
MOE: Goal

- 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?