Communication using Json

Professor Fabio Ciravegna
Department of Computer Science,
University of Sheffield
f.ciravegna@shef.ac.uk

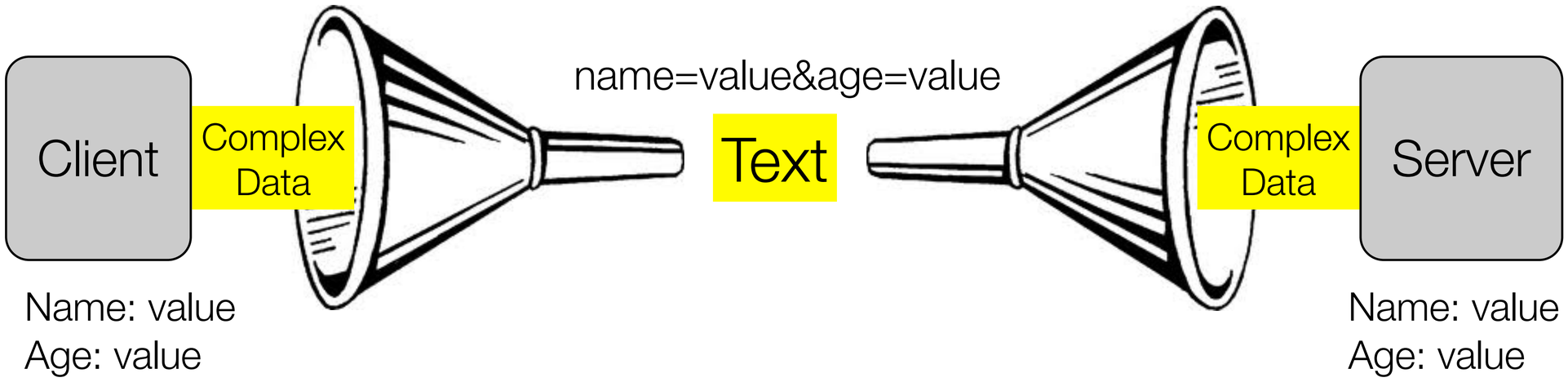
COM3504/6504
“The Intelligent Web”

http://json.org/

Client-Server communication

- Different environments
  - E.g. JavaScript on browser and server
  - Difficult communication
- Solution?
  - Serialisation/de-serialisation of data into text
  - e.g. used in HTML forms

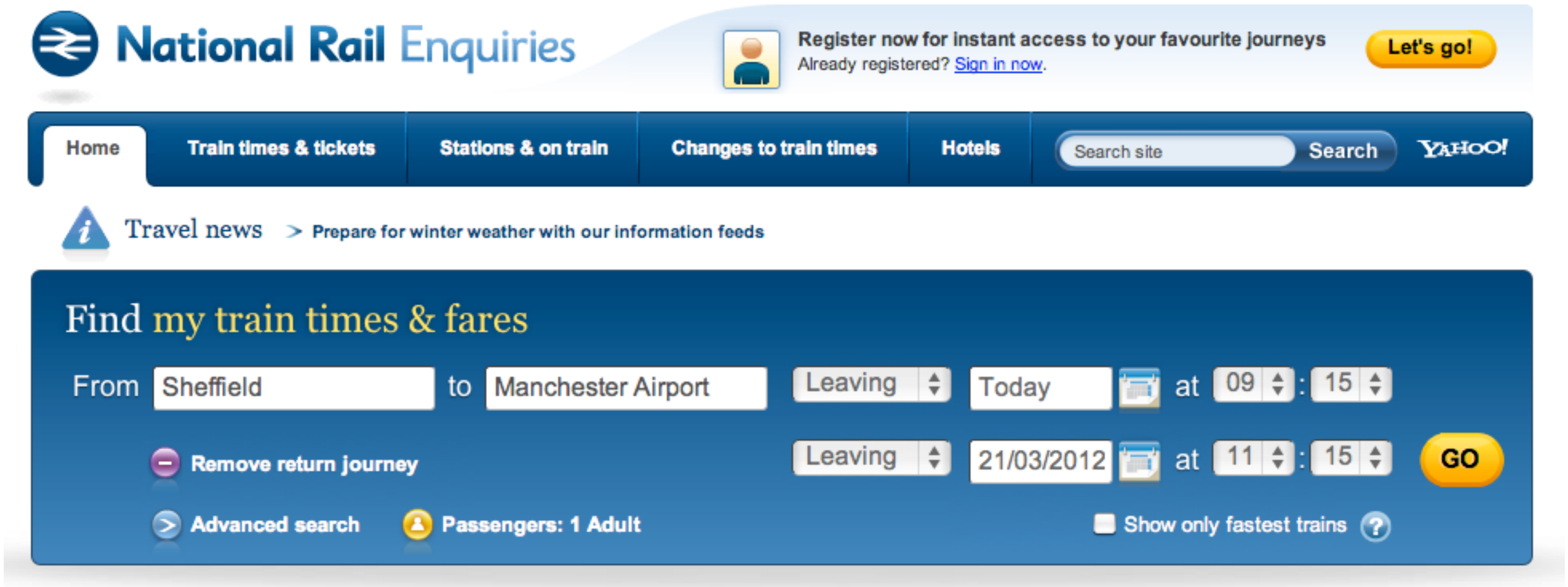
This approach is really cumbersome



The University Of Sheffield.

http://www.nationalrail.co.uk/

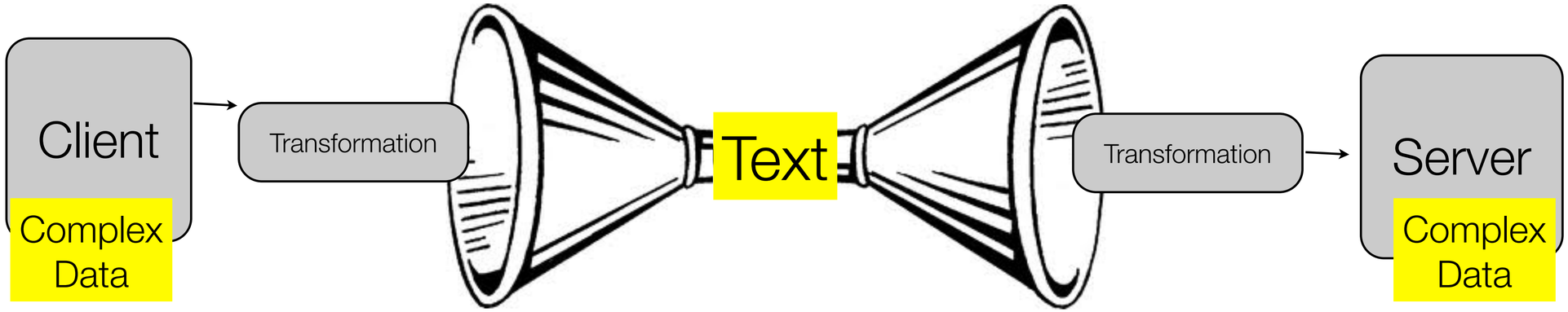
Example: National Rail Enquiry



Traditional solution

- Client and server exchange
  - Client to Server (POST)
    - `fromStation=”Shf” &toStation=”MncAir” &dateOut=”Today”&hourOut=”9” &minOur=”15” &dateIn=”21/03/2012”&….`
  - Server to Client: HTML file rendered with parameters

This approach is really cumbersome



Not because we have to turn the data into text
but because of the complex manual way we encode it
why can’t we just send the data structure?

The University Of Sheffield.

JSON

http://www.json.org/

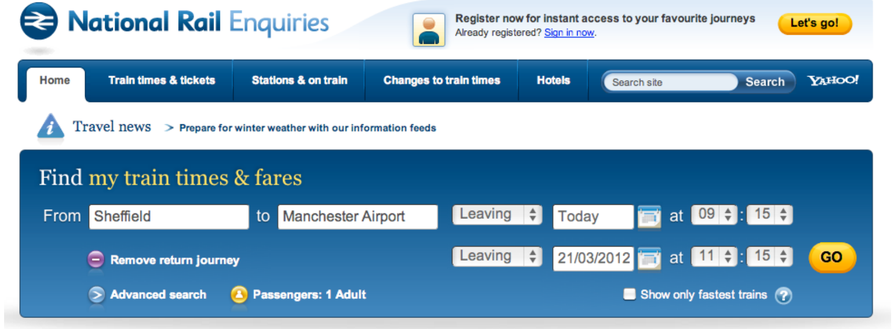
- JSON (JavaScript Object Notation) is a lightweight data-interchange format.
  - It is easy for humans to read and write. It is easy for machines to parse and generate.
  - It is based on a subset of the JavaScript Programming Language
  - JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.
  - These properties make JSON an ideal data-interchange language.

The University Of Sheffield.

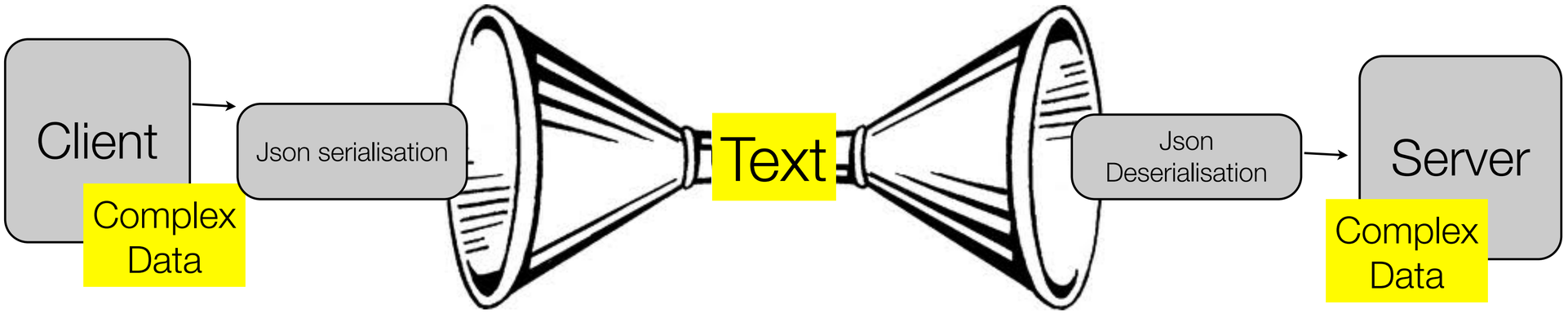
JSON (2)

- JSON is built on two structures:
  - A collection of name/value pairs. In various languages, this is realized as an
    - object, record, struct, dictionary, hash table, keyed list, or associative array
  - An ordered list of values. In most languages, this is realized as an
    - array, vector, list, or sequence.
  - These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

The University Of Sheffield.



- Client and server exchange
  - Client to Server: JSON
  - Server to Client: JSON (or HTML code)



We do not need to find a clever way to encode the data structure. Json does it for you

```
{ fromStation: 'Shf', toStation: 'MncAir', dateOut: 'Today', hourOut: '9', minOur: '15', dateIn:
```

Sender Side (Javascript)

- Serialisation

- in the context of data storage and transmission, serialization is the process of converting a data structure or object state into a format that can be stored (for example (…) transmitted across a network connection link) and "resurrected" later in the same or another computer environment (wikipedia)

```
var myJSONText = JSON.stringify(myObject, replacer);

var joe= {name: joe, height: 185, weight: 79}
var mary= {name: paula, height: 175, weight: 59}
var people= [];
people[0]= joe;
people[1]= mary;
var myJSONText = JSON.stringify(people);


>>> value of myJSONText
 `[{"name": "joe", "height":185,"weight":79},
   {"name": "paula","height":175,"weight":59}]'
```

Receiving side (Javascript)

http://www.json.org/js.html

- Deserialisation
  - The opposite operation: extracting a data structure from a series of bytes

```
var people = JSON.parse(myJSONText);
```

```
>>> people:
[{name: joe,   height: 185, weight: 79},
 {name: paula, height: 175, weight: 59}]
```

The University Of Sheffield.

Stringify additional param

- The `stringify` method can take an optional replacer function.
  - called after the toJSON method on each of the values in the structure.
  - It will be passed each key and value as parameters, and this will be bound to object holding the key.
  - The value returned will be stringified.

```
function replacer(key, value) {
    if (typeof value === 'number' && !isFinite(value)) {
        return String(value);
    }
    return value;
}
```

Receiving side (Javascript)

http://www.json.org/js.html

- Deserialisation
  - The opposite operation: extracting a data structure from a series of bytes

```
var myObject = JSON.parse(myJSONtext, reviver);
```

- The optional *reviver* function will be called for every key and value at every level of the final result.
  - Each value will be replaced by the result of the reviver function.
  - This can be used e.g. to transform date strings into Date objects.

```
myData = JSON.parse(text, function (key, value) {
    var type;
    if (value && typeof value === 'object') {
        type = value.type;
        if (typeof type === 'string' && typeof window[type] === 'function') {
            return new (window[type])(value);
        }
```

The University Of Sheffield.

Why use it?

- Client server architecture can now return data structures as opposed to HTML code
  - Client can send complex objects (as opposed to just variable-value pairs
  - Client is no longer passive: now it interprets the code and displays it as required

How to return JSON data

On a server's route:

declare that you are returning JSON in header

```
router.get('/index', function (req, res, next) {
    res.setHeader('Content-Type', 'application/json');
    res.send(JSON.stringify({ a: 1 }));
}
```

Stringify the data before sending back

For the client side we need instead to send and receive JSON instead of just getting files and posting forms

we will use AJAX (soon screened here)

Bonus Material: JSON in Java

The University Of Sheffield.

GSON

For your information

A Google library for JSON in Java

- Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of
  - Download the gson library in order to use it (it is not in the standard java distribution)

http://code.google.com/p/google-gson/

Serialisation (toGson)

For your information

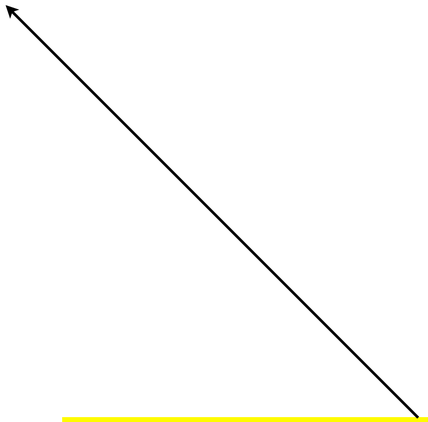
- Serialisation:

```
/* create Gson object */
Gson gson = new Gson();
/* create the object to serialise (any Java object)*/
class BagOfPrimitives {
  private int value1 = 1;
  private String value2 = "abc";
  private transient int value3 = 3;
  BagOfPrimitives() {
    // no-args constructor
  }
}
BagOfPrimitives obj = new BagOfPrimitives();
String json = gson.toJson(obj);
```

Deserialisation

For your information

```
BagOfPrimitives obj2 =
    gson.fromJson(jsonString,
                  BagOfPrimitives.class);
```



Expected Object class

Async Programming in Javascript

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

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing

The University Of Sheffield.

Whys and Whats

- Async programming is a strategy whereby the program is able to control occurrences of events outside the normal flow of the main thread
  - i.e. some code is executed when an event occurs rather than when the flow of programming requires it

- Typical example is the click of a button
  - this raises an event (buttonClick) which is intercepted by the program
    - to excuse some code

The University Of Sheffield.

<HTML>
<head>…</head>
<body>
…
<button id='button'>
…
</body>
<script>

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  alert('You clicked me!');

});
```

</script>

never insert javascript code directly into an html file, always use separate javascript files!!!

The code running the alert is run every time the button is clicked

The University Of Sheffield.

Async Javascript

- Many Web API features use asynchronous code
  - especially those that access or fetch resources from external devices,
    - e.g. files from the network, data from databases, video streams from a web cam, etc.
- You should learn how to use synch Javascript because
  - computational resources are scarce,
  - battery power is scarce
  - computing is very expensive

The University Of Sheffield.

4 ways in Js

- Async operations are stored into an event queue
- The queue is dealt with after the main thread has finished processing
  - not in parallel as Js is single threaded!!
- The queued operations are completed as soon as possible
  - They return their results to the JavaScript environment
- 4 methods
  - Callbacks (classic)
  - Timeouts and Intervals (delayed and repeated execution)
  - Promises (new)
  - Await/async (newer)

Callbacks and Associated Hell

http://callbackhell.com/

The University Of Sheffield.

What are callbacks?

- Callbacks are used in asynchronous functions
  - These are functions that take some time to execute
- They are also used in event capturing
- When you call a normal function you can use its return value immediately:

```
var result = multiplyTwoNumbers(5, 10)
console.log(result)
// 50 gets printed out
```

- But in async functions this cannot be done

```
var photo = downloadPhoto('http://coolcats.com/cat.gif')
// photo is 'undefined'!
```

- the var photo is undefined until the file has finished downloading

ctd

- So instead you store the code that you want to execute at the end of the downloading as a callback function

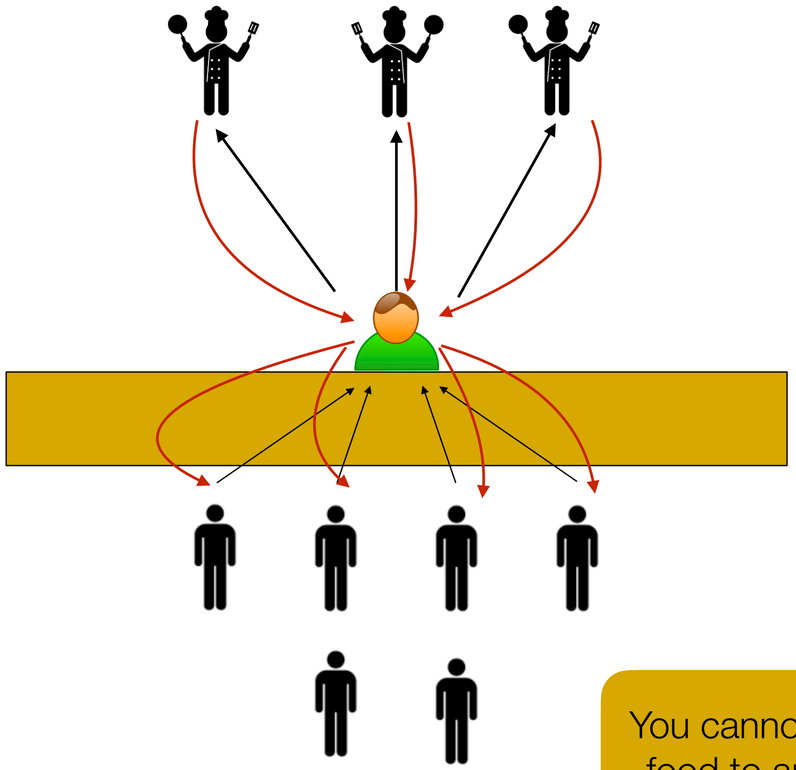
```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}

console.log('Download started')
```

- What you
  - **`Download started`**
  - …
  - **`Download finished`**
- but that is the point of async functions. They are executed later!

Event Driven and non-blocking

Remember the last lecture?

- This is very similar to a fast food outlet organisation



- Requests are posted to the till (the node.js server) which will direct them to the right cook (e.g. a database, file system, etc.)
- When a cook has prepared the food (the data), the counter (node.js server) will return it to the client
- While the food is being prepared, the till can serve other requests

You cannot start eating immediately after ordering. You must wait for the food to arrive first - equally you cannot use a photo before it is fetched

Nesting callbacks

- Issue is:
  - callbacks are often nested
  - example:
    - server is requested to get a list of images
    - actions:
      - retrieve the list of images path from the database (callback)
      - for each image path retrieve the image file (nested callback)
    - this will require nested callbacks

```
db.getImages(<fetch parameters>, function (req, res, error, data){
        for (image in data)
            fetch (image, function (req, res error, file){
                ...}
```

A very common error

- Most people will do this

This will not do what you expect (aka the code is wrong!!)

Also, I have not checked if the Javascript is completely correct - it is the spirit that is important

```
var listofphotos= [‘cat.gif’, ‘dog.gif’, ‘giraffe.gif’];
for (var ph as listofphotos)
  downloadPhoto(ph, function handlePhoto (error, photo) {
        if (error) console.error('Download error!', error)
        else photolist.push (photo);
}

zipPhotos(aCallbackofChoice, photolist);    // we suppose there is a callback
                                             // function of choice defined
```

A very common error

- Most people will do this

This will not do what you expect (aka the code is wrong!!)

```
var photolist= [];
// function that gets a list of photos taken from the file system and zips it
// and returns some metadata (e.g. average image size and number of photos) */
function zipPhotos(functionDoingStuff, photolist){
 var numOfPhotos=photolist.length;
 var totSize=0;
 for (var photo as photolist)
  totalSize+=getFileSize(photo);  // we suppose to have defined somewhere a
                                  // function getting the size of a file
 var averageSize=totalSize/numOfPhotos;
 var zippedPhotos= zipFiles(photolist); // we suppose to have defined
                                        // somewhere a
                                      // function zipping a set of files
 functionDoingStuff(null,{numOfPhotos: numOfPhotos, averageSize: averageSize
                          zippedPhotos: zippedPhotos});
 }
```

The University Of Sheffield.

Why?

- This code is wrong
  - zipphotos will receive an empty list of files
  - zipPhotos will be called before all the callbacks have finished
- Even worse:
  - this code will seem to work
  - that is because sometimes - the download of the photos is fast enough for zipPhotos to find some files in the list, while the loop in zip photos is run, some photos are added to the variable list-photos
  - So the user thinks this is working but it will be by chance: very often lots of photos will be missed.
    - It is the worst that can happen to you as a programmer:
      - an inexplicably temperamental program with bugs you are unlikely to be able to

But this is equivalent

- To go to the burger stall and thinking that you can start eating immediately after paying
  - Now, sometimes you could be able to do just that because maybe the fries are quick to come and after you have finished inputting your credit card pin code, they are already on the tray
  - So
    - you eat the fries
    - see the tray is empty
    - you leave
    - when the burger comes you have already left
      - (and you are still hungry!)
  - So remember to make sure that you wait for all your food before leaving

The University Of Sheffield.

More clever people will do this!

This will not do what you expect (aka the code is wrong!!)

```
for (var ph as listofphotos)
  downloadPhoto(ph, function handlePhoto (error, photo) {
        if (error) console.error('Download error!', error)
        else {
             photolist.push (photo);
             zipPhotos(photolist, aCallbackofChoice);
```

- This is even worse
  - Your callback will be called several times, you will get a number of zipped filed all containing one photo
    - if your callback returns to the client, it will always just return the zipped cat (and any followup callback will crash the system)

The University Of Sheffield.

So how do you do it?

- Exactly as the burger joint does
- You keep a list of all the items you have to receive
- then you keep track of the items you have received
- when you have received them all, you leave

So how do you do that?

The University Of Sheffield.

So how do you do that?

- You keep a list of all the items you have to receive
- When the callback realises that all the elements have been processed,
  - it will call the final function

```
for (var ph as listofphotos)
  downloadPhoto(ph, function handlePhoto (error, photo) {
        if (error) console.error('Download error!', error)
        else {
            photolist.push (photo);
            if (photolist.length==listofphotos.length)
               zipPhotos(photolist, aCallbackofChoice);
```

now you are sure that photos contains all the pictures required

Actually that is not correct

```
for (var ph as listofphotos)
  downloadPhoto(ph, function handlePhoto (error, photo) {
       if (error) {
            console.error('Download error!', error);
            photolist.push (null);
       } else {
            photolist.push (photo);
            if (photolist.length==listofphotos.length)
               zipPhotos(photolist, aCallbackofChoice);
```

Because otherwise, if there is an error, the server will never return

the function zipPhotos will have to filter out the null elements

And that was simple!

imagine you had to nest several layers of callback in callbacks…

```
chooseToppings(function(toppings) {
  placeOrder(toppings, function(order) {
    collectOrder(order, function(pizza) {
      eatPizza(pizza);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises

The University Of Sheffield.

How do I fix callback hell?

- Keep your code shallow

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

This code has two anonymous functions. Let's give em names!

```
var form = document.querySelector('form')
form.onsubmit = function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function postResponse (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

The University Of Sheffield.

shallow (ctd)

As you can see naming functions is super easy and has some immediate benefits:

- makes code easier to read thanks to the descriptive function names
- when exceptions happen you will get stacktraces that reference actual function names instead of "anonymous"
- allows you to move the functions and reference them by their names

Now we can move the functions to the top level of our program:

```
document.querySelector('form').onsubmit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

Always handle errors

The first two rules are primarily about making your code readable, but this one is about making your code stable. When dealing with callbacks you are by definition dealing with tasks that get dispatched, go off and do something in the background, and then complete successfully or abort due to failure. Any experienced developer will tell you that you can never know when these errors happen, so you have to plan on them always happening.

With callbacks the most popular way to handle errors is the Node.js style where the first argument to the callback is always reserved for an error.

```
var fs = require('fs')

fs.readFile('/Does/not/exist', handleFile)

function handleFile (error, file) {
  if (error) return console.error('Uhoh, there was an error', error)
  // otherwise, continue on and use `file` in your code
}
```

Having the first argument be the `error` is a simple convention that encourages you to remember to handle your errors. If it was the second argument you could write code like `function handleFile (file) { }` and more easily ignore the error.

Code linters can also be configured to help you remember to handle callback errors. The simplest one to use is called **standard**. All you have to do is run `$ standard` in your code folder and it will show you every callback in your code with an unhandled error.

Callbacks in callbacks

- The callback hell does not stop there
- You can understand the callback function and forget about it when you are inside a callback itself
- Example:
  - create a server that
    - retrieves a set of photos from another server
    - return the whole set of photos as one zipped file
- I am sure you know what to do by now
  - you will have two functions: one to retrieve the photos and one to zip them

You must understand how to get out of the callback hell

So that you can sleep peacefully at night

But you must avoid to find yourself in one at all cost

Although that is not always possible, we will now see a mechanism to avoid most (although not all) cases

Promises:
More Async Programming in Javascript

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

Promises

Escaping the callback hell

Promises

- Promises are a way to create asynchronous code that allows easy sequencing of async processes
  - Following a very familiar structure:
    - if x
      - do Y
      - then do W
      - then do Z

The University Of Sheffield.

Promise Declaration

- Step 1: declare the promise
  - the promise declares some code that is to be run asynchronously
    - for example uploading an image may take a lot of time so it is executed asynchronously
  - it also contains the strategy to decide if the promise was successfully executed or not
    - for example: it is successful if the image file was found; it is unsuccessful otherwise
  - it has two parameters:
    - function to call if code is executed successful
    - function to call if code returns an error

The University Of Sheffield.

Consuming the promise

- Here is when we run the promise so that the async code is executed
- As parameter to the call, we pass the functions to call in case of success/error
  - the success function is passed in a branch called .then, the failure function is passed in the branch called .catch
  - by passing the success/rejection functions as parameters we can use the same async code in different contexts and obtain a different behaviour
    - for example in one context the success in downloading an image will display it to the user, in another case it may compute some metadata for the image (e.g. size)
    - *so when the resolve function is called, we execute the function passed in the .then branch (or in the .catch branch otherwise)*

The University Of Sheffield.

In summary

- So the promise
  - at declaration
    - it declares a long running computation
    - it declares placeholders for the behaviour to adopt in case of success and in case of error (resolve and reject represent functions passed as parameters)
  - at consumption time
    - it declares the functions actual functions to be used for success/error
  - at execution time
    - it runs the promise code and calls the actual success/reject function

Declaration and Consumption

```
let myPromise=  new Promise(function(resolve, reject) {
 // long running operation
 let random= Math.random(20);
 // condition of success evaluated when the long running
 // operation is finished
 if (random%2===0)
        // success, so we are calling resolve
         resolve('it is even!!!');
 //failure: calling reject
 else
     reject('whoops odd number');
```

Declaration

Consumptio

```
function consumeMyPromise(){
  // calling this function will run the promis
  //(async code)
   myPromise()
     // the function to execute in
     // case of success (resolve parameter)
     .then(function(okMessage) {
        console.log(okMessage);
      })
      // the function to execute in case of
      // error (reject parameter)
    .catch(function(failureMessage)
       alert(failureMessage);
   })
```

This is just an exemplification.
Obviously the random function is not asynchronous

The Declaration Better Annotated

```
let myPromise=  new Promise(function(resolve, reject) {

 let random= Math.random(20);

 if (random%2===0)

        resolve('it is even!!!');

 else
      reject('whoops odd number');
```

Declaration

Code that takes a long time

Condition of success/error

call to the success function

call to the success function

Example

```
myPromise()
.then(function(okMessage) {
   console.log(okMessage);
   })
 .catch(function(failureMessage)
  alert(failureMessage);
  })
```

or - if you prefer using the fat arrow notation…

```
myPromise()
.then(okMessage =>
   console.log(okMessage);
   )
 .catch(failureMessage =>
  alert(failureMessage);
  )
```

- Note: if you want to use the fat arrow notation (ES6) in IntelliJ you must
- enable ES6
  - https://hackernoon.com/quickstart-guide-to-using-es6-with-babel-node-and-intellij-a83670afbc49
  - read until the math example, the rest is irrelevant

Concatenating Promises

- Promises gets you out of the callback hell
- Instead of nesting callbacks, you just concatenate .then branches

```
function orderFood(){
 chooseToppings(function(err, toppings){
   if (err) {
      alert (err);
      return;
      }
   placeOrder(toppings, function(err, order){
       if (err) {
         alert (err);
        return;
        }
        collectOrder(order,function(err, pizza) {
                  if (err) {
                   alert (err);
                   return;
                  }
                 eatPizza(pizza, function()
        return;);})})})})})
```

Vs

```
chooseToppings()
.then(toppings =>
  placeOrder(toppings))
.then(order =>
  collectOrder(order))
.then(pizza =>
  eatPizza(pizza))
.catch(err =>
      alert (err);
);
```

Passing values from the environment

- you can pass values to a promise in this way
  - you include the promise in a function with parameters
  - use the name of the parameters within the promise

```
var loginF = function(username, password) {
  return new Promise(function(resolve, reject) {
    if (correct(username, password) {
      resolve("login accepted");
    } else {
      reject(Error(“Login failed"));
    }
  });
}
```

https://stackoverflow.com/questions/35318442/how-to-pass-parameter-to-a-promise-function

Consuming Promises with parameters

- So when you consume the promise:

```
loginF(username, password)
 .then(function(uid) {
    // show login page
  }
  .catch (error){
   // show failure message
  })
```

The University Of Sheffield.

What you should know

- You should be confident in writing async programmes in Javascript using promises
  - you should avoid having nested callbacks when possible
- You should be able to:
  - Declare a promise
  - Consume a promise
  - Pass parameters to a promise

Questions?

Asynchronous Computing using Async/Await

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

- Async functions and the await keyword (ECMAScript 2017) are **syntactic sugar** on top of promises,
  - making asynchronous code easier to write and to read afterwards
    - but it is still asynchronous!
  - They make async code look more like old-school synchronous code

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await

Async keyword

- It takes a normal function and returns a promise

```
function hello() { return "Hello" };

hello();
```

```
async function hello() { return "Hello" };
(or)
let hello = async () => { return "Hello" };
```

```
hello().then((value) => console.log(value))
```

The University Of Sheffield.

Await

- putting away in front of a call to an async function, allows to stop the flow of the code until the promise is fulfilled or rejected
  - very useful as the Js thread can execute some other code / react to other events if necessary

```
fetch('coffee.jpg')
.then(response => response.blob())
.then(myBlob => {
  let objectURL = URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
})
.catch(e => {
  console.log('There has been a problem with your fetch operation: ' + e.message);
});
```


```
async function myFetch() {
  let response = await fetch('coffee.jpg');
  let myBlob = await response.blob();

  let objectURL = URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}


// when we call it we must take care of any error that any promise in the
// body will throw
myFetch()
.catch(e => {
  console.log('There has been a problem with your fetch operation: ' + e.message);
});
```

The University Of Sheffield.

Error handling

- In case of error, you can:
  - return a condition of error intercepted by the calling function
    - e.g. a specific value
  - throw an error
    - in that case it can be captured by the calling function

The downsides of async/await

- Async/await makes your code look synchronous,
  - and in a way it makes it behave more synchronously
- The await keyword blocks execution of all the code that follows until the promise fulfils
  - exactly as it would with a synchronous operation
  - It does allow other tasks to continue to run in the meantime, but your own code is blocked

Promise.all

- Your code could be slowed down by a significant number of awaited promises
  - Each await will wait for the previous one to finish
    - Whereas sometimes you have tasks that can easily run in parallel
      - you may want the promises to begin processing simultaneously, like they would do if we weren't using async/await
      - also you may want that when one fails, all the others are not completed
  - Promise.all does all this
    - promises run in parallel
    - if one fails, the other fail as well
    - one point of error

Example

```
async function delayAll() => {
  await Promise.all([
    Promise.delay(600),
    Promise.delay(600),
    Promise.delay(600)]); //runs all delays simultaneously
};
```

https://stackoverflow.com/questions/45285129/any-difference-between-await-promise-all-and-multiple-await

```
delayAll()
  .then(function(arrayOfValuesOrErrors) {
     // handling ok case
  })
  .catch(function(err) {
    console.log(err.message); // some coding error in handling happened
  });
```

https://stackoverflow.com/questions/30362733/handling-errors-in-promise-all

The University Of Sheffield.

Errors

- Error handling must be implemented using the reject branch of each promise and then collected with catch by the calling function
  - only if failure in one causes failure in all the others
  - if failure in one is not relevant to the others, then do not reject.
    - just resolve with a form of null value

What you have learned

- You should be confident in using async/await as a way to simplify writing code involving promises
- In week 4 we will use the construct considerably

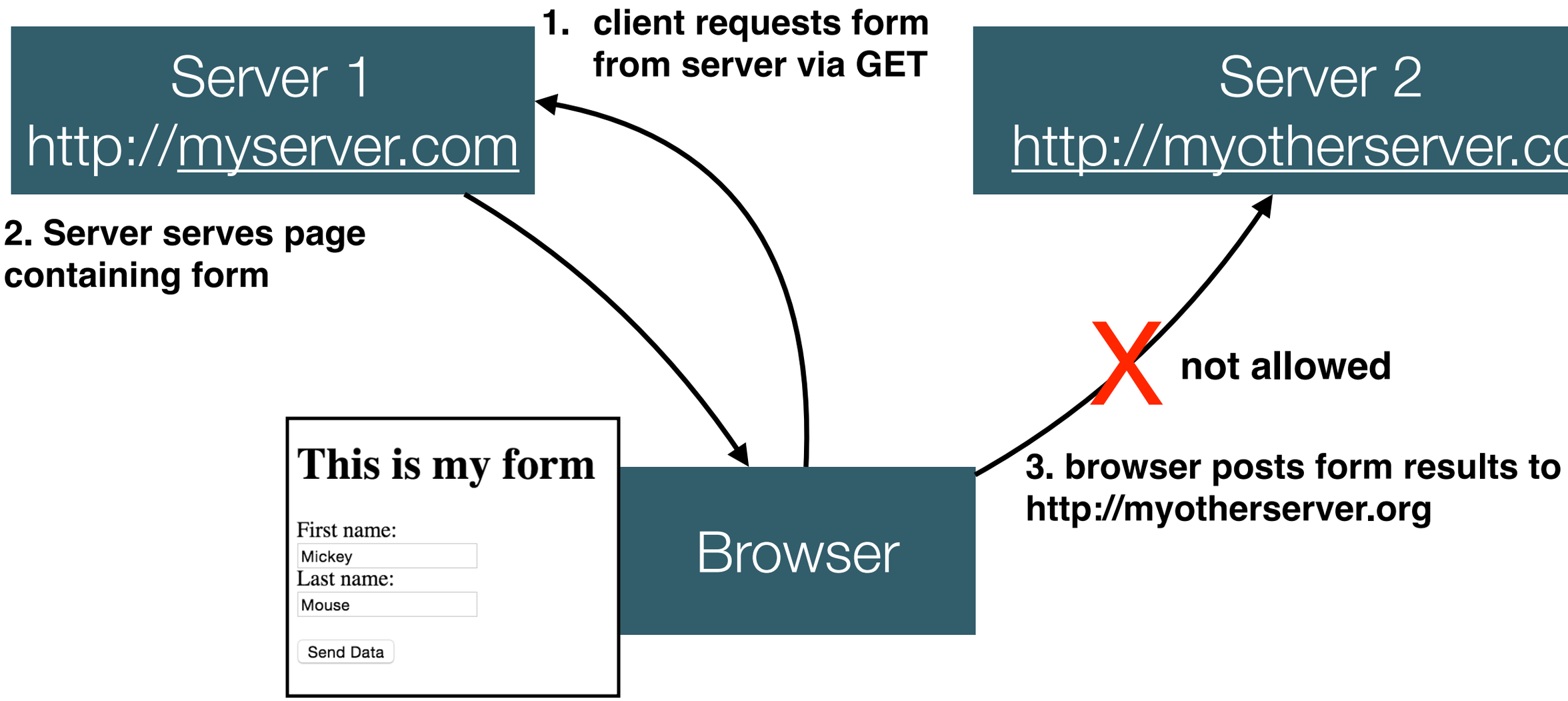
The University Of Sheffield.

A common Issue: CORS (when the client is a browser)

Professor Fabio Ciravegna
Department of Computer Science,
University of Sheffield
f.ciravegna@shef.ac.uk

COM3504/6504
“The Intelligent Web”

Beware! CORS



If the server sending the page is **http://myserver.org:63342,** you are not allowed to post to another server (e.g. http://myotherserever.org:3000). This is to avoid man in the middle attacks. You must post to the same server. You are not even allowed to post the the same server on another port. Origin '**http://myserver.org:63342**' is therefore not even allowed posting to '**http://myserver.org**:3000'.

Beware!

- you must post to the same server serving the html file (including port!) otherwise the browser will refuse to send your request
- To avoid this block, the server *must* declare that the page is allowed to post elsewhere,

i.e. the server serving the html file you must set

- response.writeHead(200,
  {(…)
  '**Access-Control-Allow-Origin' : '**http://myotherserver.org:3000'
  });

See also Cross-origin resource sharing: a simple method to perform cross-domain requests by introducing a small proxy server able to query outside the current domain
There is a simple way of doing it in node.js, php, etc.

CORS! The right way

1. client requests form from server via GET

Server 1
http://myserver.com

Server 2
http://myotherserver.com

2. Server serves page containing form and specifying

response.writeHead(200,
{(…)
'Access-Control-Allow-Origin' :
‘http://myotherserver.com});

allowed!

Browser

This is my form

First name:
Mickey
Last name:
Mouse

Send Data

form posts to
http://myotherserver.com/

Never ever ever return
'Access-Control-Allow-Origin' : ‘*’
as it opens your server to man in the middle attacks

The University Of Sheffield.

Why?

http://stackoverflow.com/questions/10636611/how-does-access-control-allow-origin-header-work

Access-Control-Allow-Origin is a CORS (Cross-Origin Resource Sharing) header.

When Site A tries to fetch content from Site B, Site B can send an Access-Control-Allow-Origin response header to tell the browser that the content of this page is accessible to certain origins. (An origin is a domain, plus a scheme and port number.) By default, Site B's pages are not accessible to any other origin; using the Access-Control-Allow-Origin header opens a door for cross-origin access by specific requesting origins.

For each resource/page that Site B wants to make accessible to Site A, Site B should serve its pages with the response header:

Access-Control-Allow-Origin: http://siteA.com
Modern browsers will not block cross-domain requests outright. If Site A requests a page from Site B, the browser will actually fetch the requested page on the network level and check if the response headers list Site A as a permitted requester domain. If Site B has not indicated that Site A is allowed to access this page, the browser will trigger the XMLHttpRequest's error event and deny the response data to the requesting JavaScript code.

Note

- Please do not confuse posting (i.e. sending data) and getting (retrieving a file)
  - of course you can have gets pointing to different servers in any html file
  - e.g. this is allowed

```
<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.j
    </script>
</head>
```

- Instead the client browser cannot **POST** data elsewhere without being allowed to do so explicitly by the server
  - so that your personal data is not sent to unauthorised people

The University Of Sheffield.

Questions?

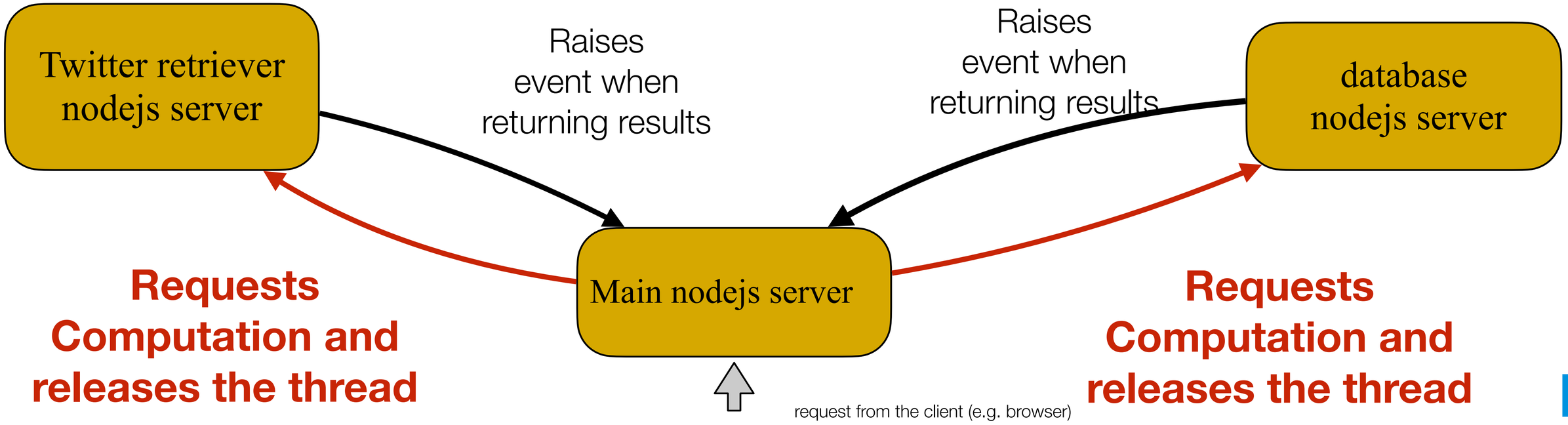
Server to Server Communication

Professor Fabio Ciravegna
Department of Computer Science
University of Sheffield
f.ciravegna@shef.ac.uk
http://staffwww.dcs.shef.ac.uk/people/F.Ciravegna/

NodeJs not to be used for computationally intensive tasks

From week 1

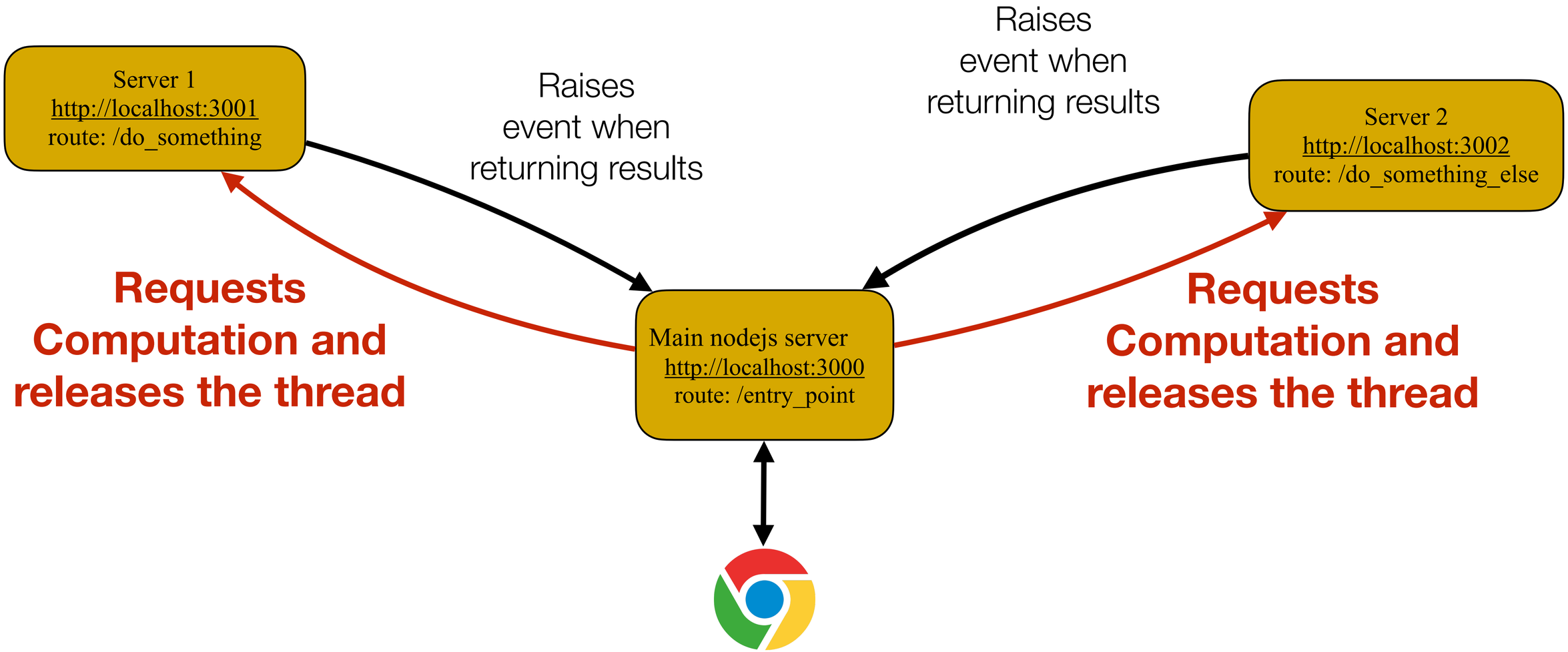
- Organise your server so that the main loop (capturing the http/s request event) is never blocked by heavy computation
  - Use a small constellation of fast specialised nodejs servers around it doing the computation
- Today we are going to learn to do that



Posting/Getting requests from node.js

- To implement that we need to be able to create communication among servers
  - Solution:
    - we will create different servers (i.e. new IntelliJ projects)
    - each server will run on a different port of localhost and have its own routes
    - some of these routes will POST/GET to/from the routes of the other servers

An example



How to implement it

- Server 1 and 2 will not be different from the servers we have seen so far
  - e.g.

```
router.post('/message', function(req, res, next) {
    res.end(JSON.stringify({message: hello});
});
```

  - but the main server will have to do something like

```
router.get('/whatever', function(req, res, next) {
    //  here we should post to the other server and get the result
    // something like:
        contactOtherServer(function(result)
            res.end(JSON.stringify(result)
        })
});
```

the actual code is in the next slide

https://www.npmjs.com/package/node-fetch

The node-fetch library

`to install: open the terminal window in IntelliJ (bottom left) and type  npm install node-fetch`

```
const fetch = require('node-fetch');

let whatever= req.body.whatever; // whatever we receive from the browser
// Set the headers
let headers = {
        method: 'post',
        body:    JSON.stringify(whatever),
        headers: { 'Content-Type': 'application/json' },
        user-agent: 'localhost:3000'
    })


fetch('http://localhost:3001/do_something', headers)
    .then(res => res.json()) // expecting a json response e.g. {field1: 'xxx', field 2: 'yyy'}
    .then(json =>
      res.render('index', {title: " results is: "+json.field2}))
    .catch(err =>
      res.render('index', {title: err}))
```

HTTP headers

Parameters for the POST (we suppose there is a variable whatever received from somewhere)

res.render shows the ejs file

e.g. we display the value of field2 in the title

if there is an error, as a result we display the same index file with the error in the title

How to send a POST from a node.js server to another server

Make sure to check the npm page

e.g. to know how to perform a get and to now more details

https://www.npmjs.com/package/node-fetch

Common Usage

NOTE: The documentation below is up-to-date with `2.x` releases; see the `1.x` readme, changelog and 2.x upgrade guide for the differences.

Plain text or HTML

```
fetch('https://github.com/')
    .then(res => res.text())
    .then(body => console.log(body));
```

JSON

```
fetch('https://api.github.com/users/github')
    .then(res => res.json())
    .then(json => console.log(json));
```

Simple Post

```
fetch('https://httpbin.org/post', { method: 'POST', body: 'a=1' })
    .then(res => res.json()) // expecting a json response
    .then(json => console.log(json));
```

Post with JSON

```
const body = { a: 1 };

fetch('https://httpbin.org/post', {
        method: 'post',
        body:    JSON.stringify(body),
        headers: { 'Content-Type': 'application/json' },
    })
    .then(res => res.json())
    .then(json => console.log(json));
```

Post with form parameters

`URLSearchParams` is available in Node.js as of v7.5.0. See official documentation for more usage methods.

NOTE: The `Content-Type` header is only set automatically to `x-www-form-urlencoded` when an instance of `URLSearchParams` is given as such:

```
const { URLSearchParams } = require('url');

const params = new URLSearchParams();
params.append('a', 1);

fetch('https://httpbin.org/post', { method: 'POST', body: params })
    .then(res => res.json())
    .then(json => console.log(json));
```

Handling exceptions

NOTE: 3xx-5xx responses are *NOT* exceptions and should be handled in `then()`; see the next section for more information.

Adding a catch to the fetch promise chain will catch *all* exceptions, such as errors originating from node core libraries, network errors and operational errors, which are instances of

Connecting node.js to MySQL

Just in case you ever need it

we use it as an example of a streaming api like the one used in Twitter

A streaming API is one that sends data at intervals

The University Of Sheffield.

http://www.codediesel.com/nodejs/querying-mysql-with-node-js/

- Download the package
  - npm install mysql
- Modify your server to query the database
- Send query
- Read results as row[i].*field_name*

you must run *npm install mysql*

```
var mysql = require('mysql');
 … (insert app.post here or whatever you need)
var connection = mysql.createConnection(
    {
      host     : 'your_mysql_server',
      port     : '3306',
      user     : 'your-username',
      password : 'your-password',
      database : 'your_db_name',
    }
);
connection.connect();

var queryString = 'SELECT * FROM your_relation';
connection.query(queryString,
 function(err, rows, fields) {
     if (err) throw err;
     for (var i in rows)
    console.log('name: ' + rows[i].name +
     ' ', rows[i].surName);
});
connection.end();
```

Callback function (called when results are received)
- err: contains an error if any
- rows is an array of database records
- fields are the available fields in the records (i.e. names of columns)

Processing data while it arrives

- The previous example collects all the data and then, when finished, it processes it
  - it may be very inefficient (and go out of memory) if results are very large
- It is possible to process data while it arrives using events
  - This is a typical software pattern in node.js

While it arrives

```
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

The University Of Sheffield.

What you should know

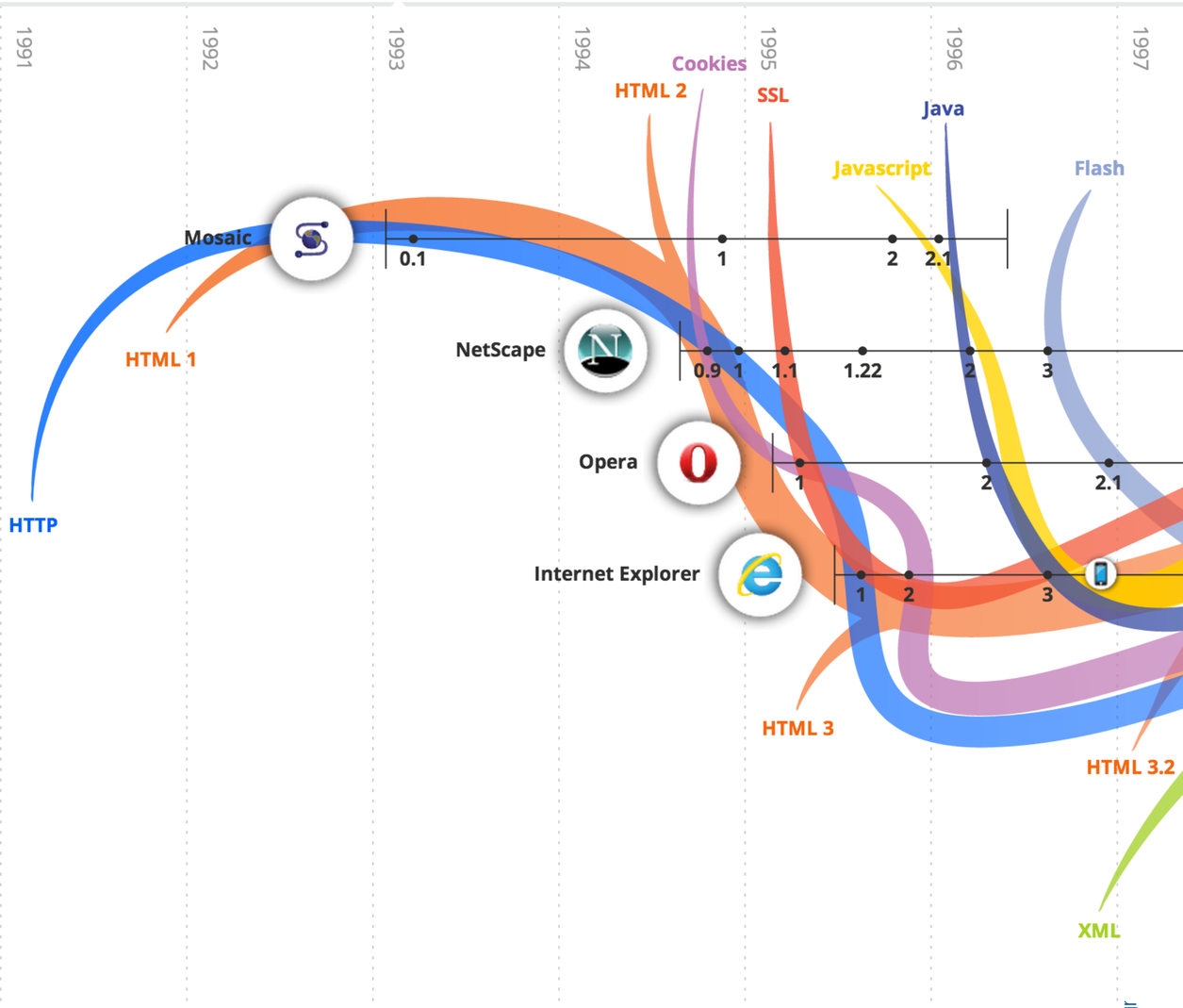
- You should understand why you need a constellation of servers
- You should know how to create multiple servers in IntelliJ
- You should know how to connect one server to the other via the fetch library
  - sending data
  - receiving data
- Be aware that some APIs require the sending and receiving of large data sets
  - so they allow to receive the data in packets
  - we will not work with these but it is important to remember that they exist

Thank you

The University Of Sheffield.

The Http Protocol

Professor Fabio Ciravegna
University of Sheffield



Credits

The 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.

The University Of Sheffield.

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

The University Of Sheffield.

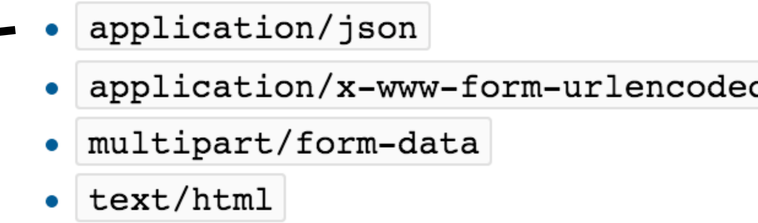
HTTP: GET and POST methods

- 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.



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

The University Of Sheffield.

Before we continue...

- Please consider that the HTTP protocol is very simple but the interaction between client and server must be done in a precise way, passing
  - As much information as possible
    - To be understood
    - To generate only the necessary traffic and no more
      - Do not ask for things you do not need
      - You are not the one paying for the server, the provider is
      - Although you both pay for the bandwidth;
  - The correct identification information about the client

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!

The University Of Sheffield.

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

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

Responses Codes

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.

The University Of Sheffield.

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)

The University Of Sheffield.

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/

The University Of Sheffield.

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

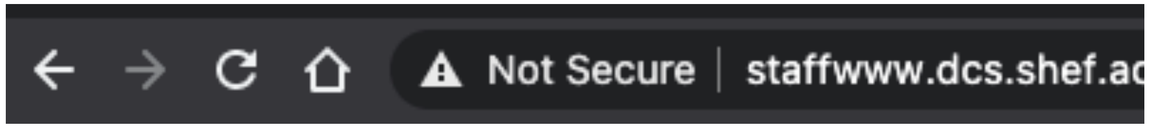
A council put up a Welsh language road sign reading "I am out of the office at the moment" when it should have said "No entry for heavy goods vehicles".

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

Internet

What is HTTP/2 and is it going to speed up the web?

Biggest change to how the web works since 1999 should make browsing on desktop and mobile faster

Samuel Gibbs

@SamuelGibbs

Wednesday 18 February 2015 15.10 GMT

The internet is set to get quicker as the biggest change to the protocols that run the web since 1999 arrives with HTTP/2. Photograph: Alamy

The University Of Sheffield.

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

The University Of Sheffield.

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?