



The
University
Of
Sheffield.

Ajax, Json, Promises and More Express

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

COM3504/6504
“The Intelligent Web”

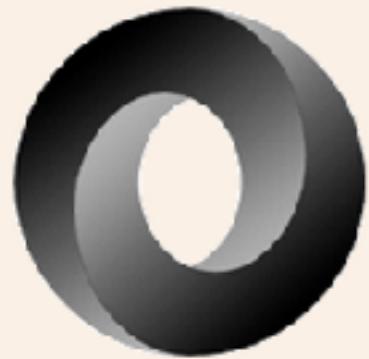


The
University
Of
Sheffield.

Exchanging data

JSON

<http://json.org/>

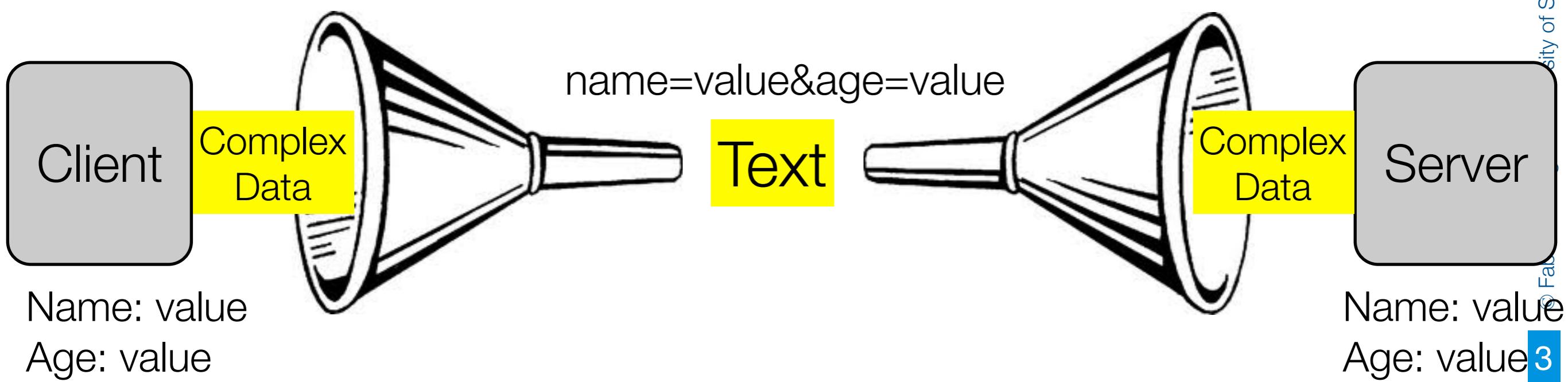


Introducing JSON

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

 **National Rail Enquiries**

 Register now for instant access to your favourite journeys
Already registered? [Sign in now](#).

[Let's go!](#)

[Home](#) [Train times & tickets](#) [Stations & on train](#) [Changes to train times](#) [Hotels](#) [Search](#) [YAHOO!](#)

 Travel news > Prepare for winter weather with our information feeds

Find my train times & fares

From to Leaving Today  at :

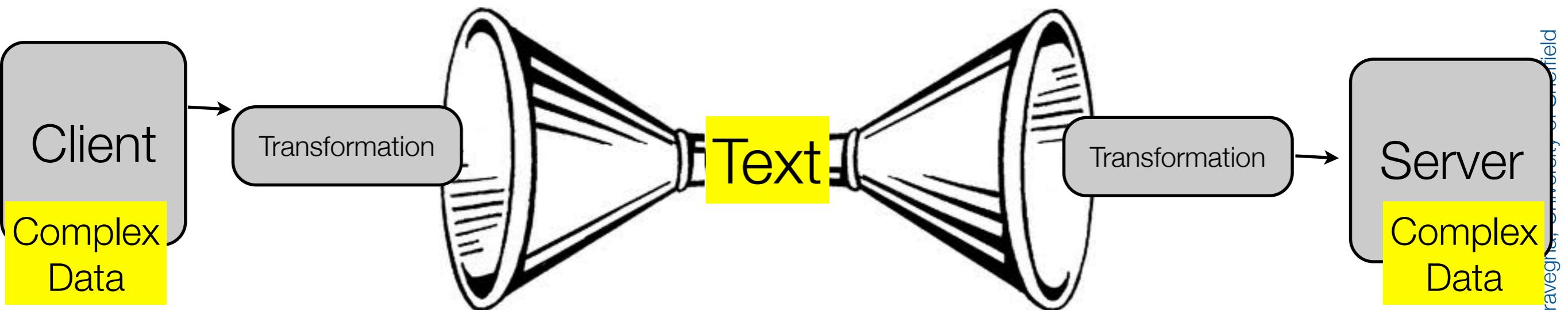
Remove return journey

Leaving  at : [GO](#)

Advanced search  Passengers: 1 Adult Show only fastest trains 

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?



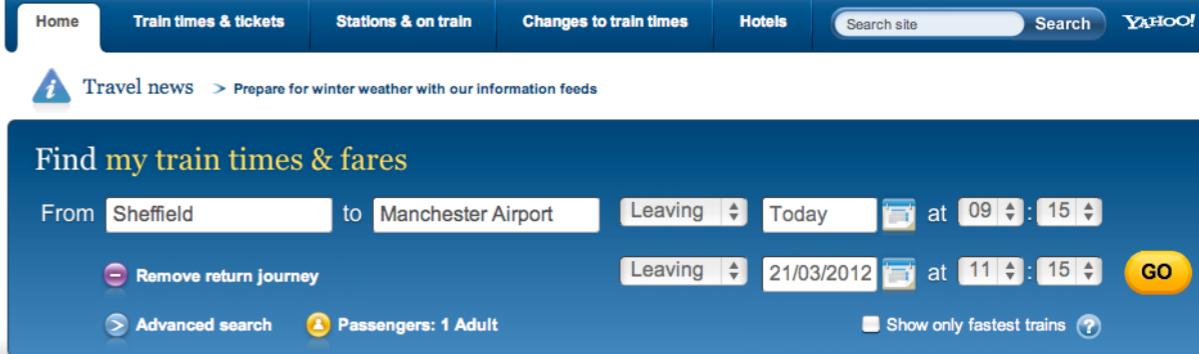
JSON

<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.

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.

Travel news > Prepare for winter weather with our information feeds

Find my train times & fares

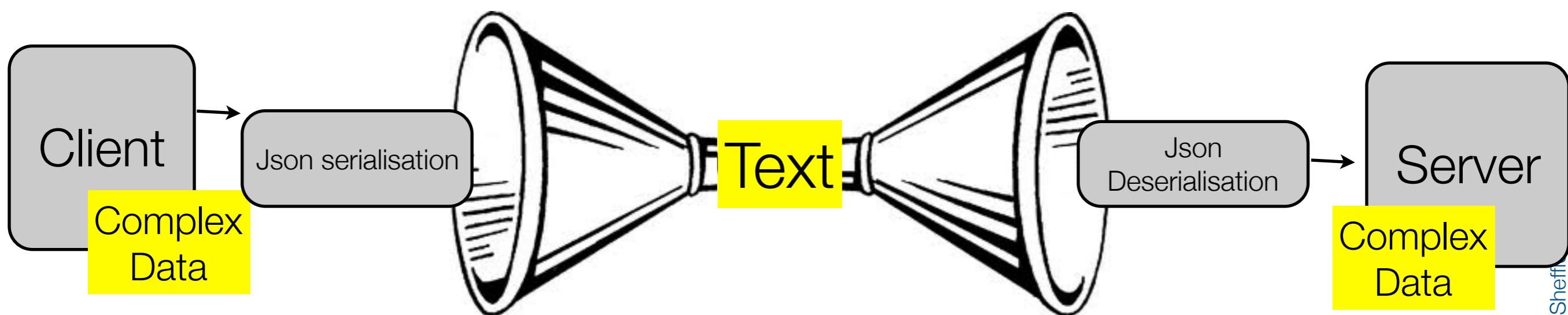
From to Leaving at

Leaving at

Passengers: 1 Adult

- 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',
minOut: '15', dateIn: '21/03/2012' }
```

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} ]
```

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);  
        }  
    }  
});
```

JSON

A Google library for JSON in Java

For your information

- 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)

- 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

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

For your information

Expected Object class

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:

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

declare that you are returning JSON
in header

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)



The
University
Of
Sheffield.

Async Programming

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

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 execute some code

```
<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

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

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)



The
University
Of
Sheffield.

Callbacks and Associated Hell

<http://callbackhell.com/>

What are callbacks?

- Callbacks are used in asynchronous functions
 - These are functions that take some times to execute
 - 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

- 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')
```

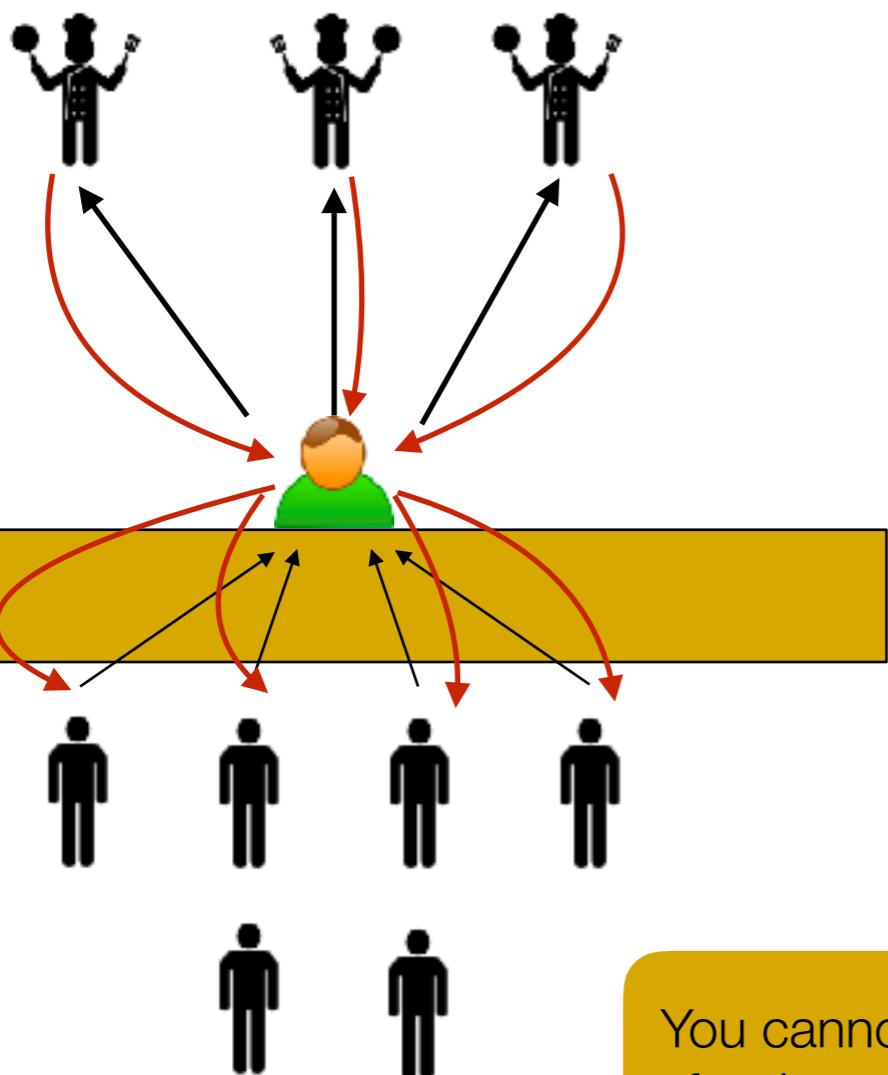
- What you see on the log on console may seem strange:
 - **Download started**
 - ...
 - **Download finished**
- but that is the point of async functions. They are executed later!



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

```
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});
}
```

Why?

- This code is wrong
 - zippotos 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 recreate in testing conditions

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

More clever people will

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

```
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 files all containing one photo
 - if your callback returns to the client, it will always just return the zipped cat

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



The
University
Of
Sheffield.

McDonald's Restaurant #4525
NORTH 3416 MARKET
SPOKANE, WA 99207
TEL# 509 489 9723

KS# 3
Side1

09/12/2017 08:42 AM
Order 75



1 Egg Cheese Biscuit	2.69
1 L Coke	1.00

Subtotal	3.69
Tax	0.32
Eat-In Total	4.01

So how do you do it?

- 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!', err
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);
```



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
  })
}
```



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



The
University
Of
Sheffield.

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



The
University
Of
Sheffield.

Promises

Escaping the callback hell

<https://scotch.io/tutorials/javascript-promises-for-dummies>

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

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
 - where
 - every **.then** branch is called when the results of the previous branch is returned
 - every part of this code is asynchronous
 - every synch part if executed in sequence

Or if you prefer

- Promises allow you to defer actions
 - until after a previous action has completed, OR
 - to respond to its failure
- A Promise is an object that represents an intermediate state of an operation
 - A promise that a result of some kind will be returned at some point in the future
- There is no guarantee of exactly when the operation will complete and the result will be returned,
 - but there is a guarantee that when the result is available, or the promise fails, the code you provide will be executed

```
chooseToppings()
  .then(function(toppings) {
    return placeOrder(toppings); })
  .then(function(order) {
    return collectOrder(order); })
  .then(function(pizza) {
    eatPizza(pizza); })
  .catch(failureCallback);
```

or - if you prefer...

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

Promises

<https://scotch.io/tutorials/javascript-promises-for-dummies>

- Imagine you are a kid. Your mom promises you that she'll get you a new phone next week.
 - You don't know if you will get that phone until next week.
 - Your mom can either really buy you a brand new phone
 - Or stand you up and withhold the phone if she is not happy
- A promise has 3 states:
 - Pending:
 - You don't know if you will get that phone until next week.
 - Resolved:
 - Your mom really buys you a brand new phone.
 - Rejected:
 - You don't get a new phone because your mom is not happy

Declaring

```
let promiseFetchImage = fetch('coffee.jpg');
```

```
let promiseCreateBlob = promiseFetchImage
    .then(response => response.blob());
```

(fat arrow notation: equivalent to)

```
let promiseCreateBlob = promiseFetchImage
    .then(function(response){
        return(response.blob())));
```

```
let promiseShowImage = promiseCreateBlob.then(myBlob => {
    let objectURL = URL.createObjectURL(myBlob);
    let image = document.createElement('img');
    image.src = objectURL;
    document.body.appendChild(image);

})
```



- All together now!!!

```
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);
  });

```

.then= status: resolved

.catch= status: failed

- Note however that we have just consumed promises, i.e. used a function (fetch) that already declares a promise



Declaring and Consuming

- There are two moments:
 - the declaration that is a generic statement of a promise
 - it shows intention but the promise is not operational
 - The use (consuming) of the promise
 - it activates the promise
 - **Note: the code will not be execute immediately**
 - **it is a callback waiting for an event to fire asynchronously**

Declaring

- It defines the behaviour of the callback
- it defines reasons of success or failure
- it defines what data to return in case of failure or success

Note the declaration is generic: it can be reused in different contexts where the actual action is different

for example a promise to find a named file can be used both for returning the file and for deleting it

Consuming

- It starts the listener for the callback event
- it allows concatenating promises
- it intercepts success/failure and will take appropriate actions (i.e. here we specify what to do with the data retrieved by the promise)

In the file example, here we state if the file is to be deleted or returned

Creating Promises

- It is possible to build your own promises using the `Promise()` constructor
 - The main situation in which you'll want to do this is when you've got code based on an old-school asynchronous API that is not promise-based, which you want to promis-ify.
 - This comes in handy when you need to use existing, older project code, libraries, or frameworks along with modern promise-based code

- A promise has two parameters:
 - resolve and reject
 - these are **functions** that you call to fulfil or reject the newly-created promise
 - resolve will fire the .then branch
 - reject will fire the .catch branch of the promise

```
let promise= new Promise(function(resolve, reject) {  
    let random= Math.random(20);  
    if (random%2==0)  
        resolve('it is even!!!');  
    else {  
        reject('whoops odd number');  
    }  
})
```

Passing parameters to Promises

- Promises get values from the environment

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

Consuming Promises

Consuming

- So far we have just defined the behaviour of the promise.
 - We have not started the counter to the day we will get the phone
- Consuming a promise means to activate them,
 - i.e. when we fire the rule and check if mum is happy and therefore we will get the phone

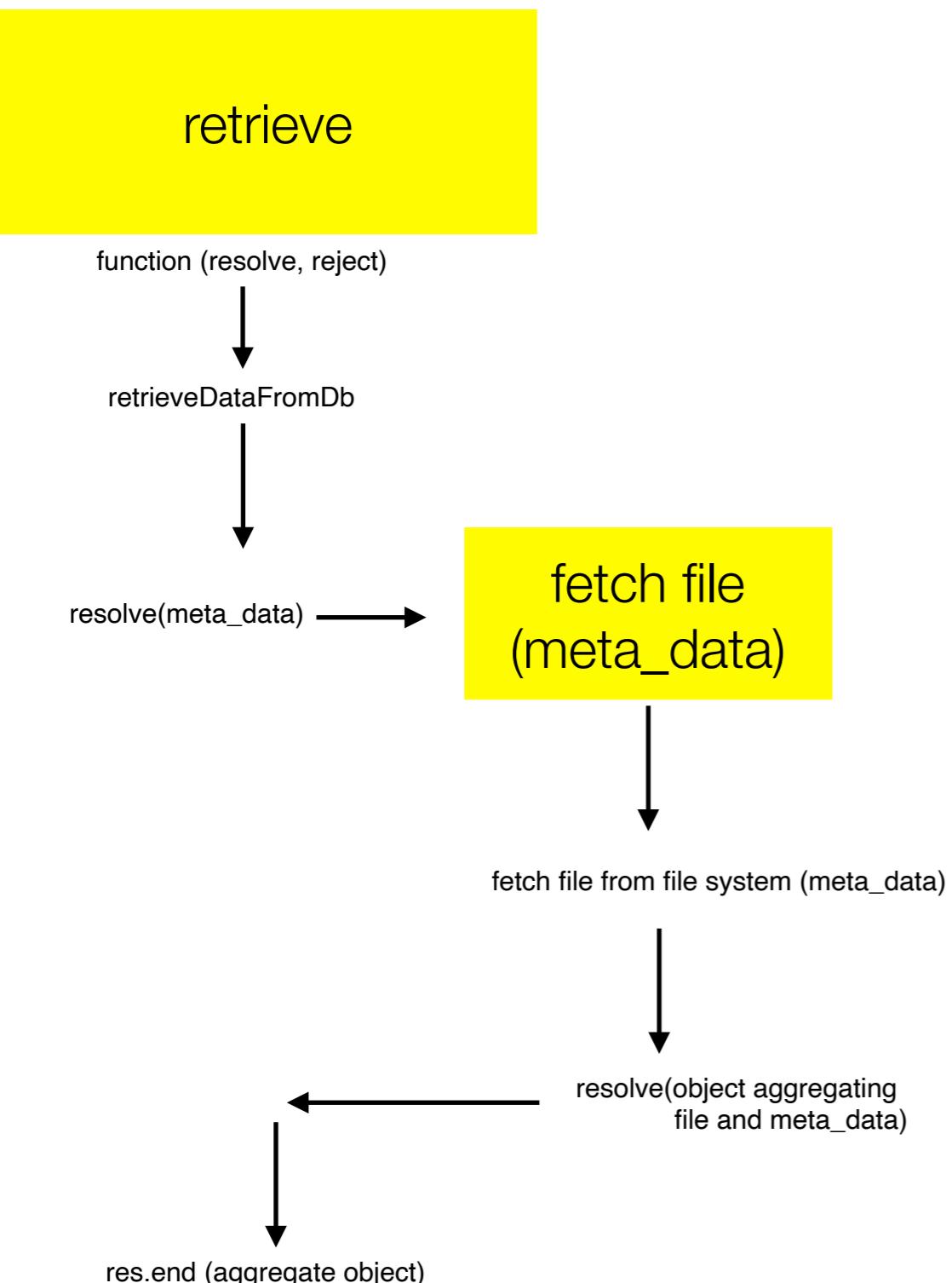
```
loginF(username, password)
  .then(function(uid) {
    // show login page
  })
  .catch (error){
    // show failure message
  })
```

ES6 / ES2015

- Different versions of Javascript have improved expressivity
 - ES6 supports promises natively.
 - In addition we can further simplify the code with fat arrow
=>
 - and use **const** (constants) and **let** (variables local to the block)
- Note: if you want to use 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

A better example

- retrieve file metadata from database
 - e.g. path, date, authors, annotations, etc.
 - fetch the file itself
 - return file



```
var promise1 = new Promise(function (resolve, reject) {
  var meta_data = retrieveDataFromDb(req.data);
  if (meta_data)
    resolve(meta_data);
  else reject();
});

var promise2 = function (meta_data) {
  if (meta_data) {
    return new Promise(function (resolve, reject) {
      var file = retrieveFileFromFileSystem(meta_data);
      resolve({file: file, meta_data: meta_data});
    });
  } else {
    reject();
  }
};

//actuation
promise1
  .then(promise2)
  .then(function (aggregateObject) {
    res.end(JSON.stringify(aggregate));
  })
  .catch(function (error) {
    res.status(404).end();
  });

```

initial promise: fetch metadata and return it to the next `.then` branch

it receives the `meta_data` which e.g. contains the file path in the file system
it retrieves it and returns both the `meta_data` and the file to the next `.then` branch

it returns the aggregate data to the client

this is part of a route, so `res` is visible



The
University
Of
Sheffield.

Async and await

- 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.
 - 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))
```

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
- Remember this code?

```
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 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 fulfills
 - 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
- Your code could be slowed down by a significant number of awaited promises
 - Each await will wait for the previous one to finish
 - Whereas actually what you want is for the promises to begin processing simultaneously, like they would do if we weren't using async/await



The
University
Of
Sheffield.

Ajax

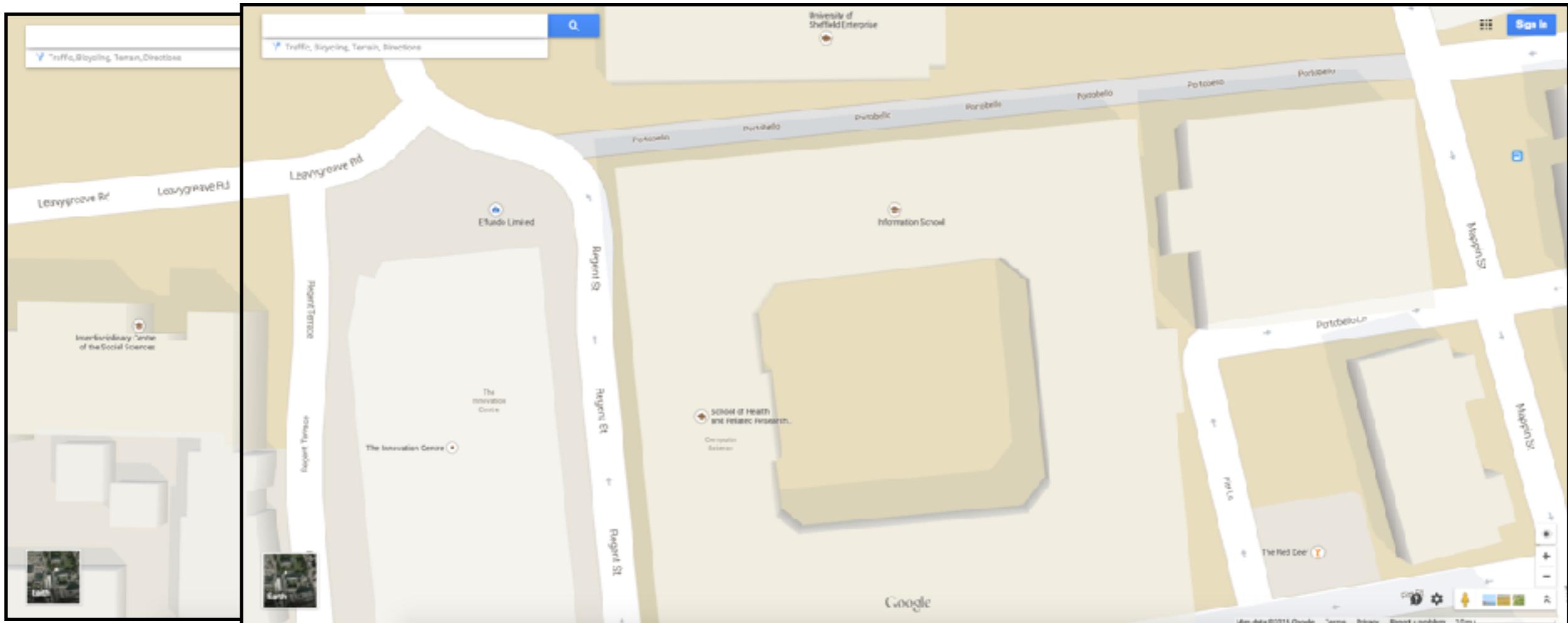
Asynchronous client-server communication

The role of 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

The traditional Web

- Http protocol is memoryless
- you have to send a complete request every time



- If I want to move towards Mapping Street, typically I would have to send a request for a complete map



Towards a different Client-Server

- 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?

Ajax: A New Approach to Web Applications

by [Jesse James Garrett](#)

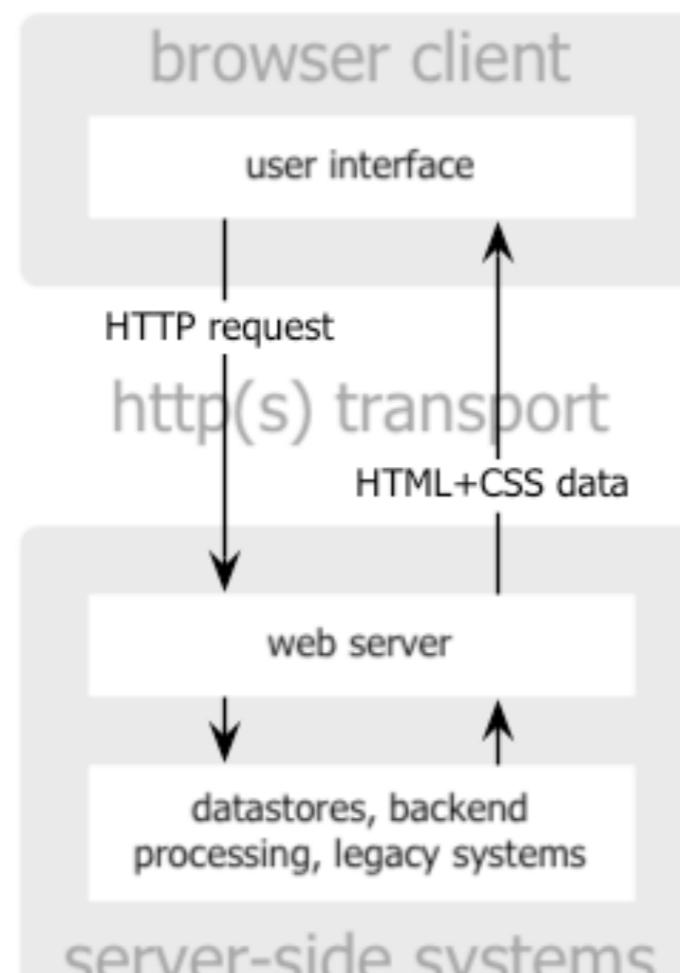
<http://adaptivepath.com/publications/essays/archives/000385.php>

As we have seen HTTP2 is redesigning the web so that the server and the client cooperate to reduce traffic and improve speed

Ajax is different

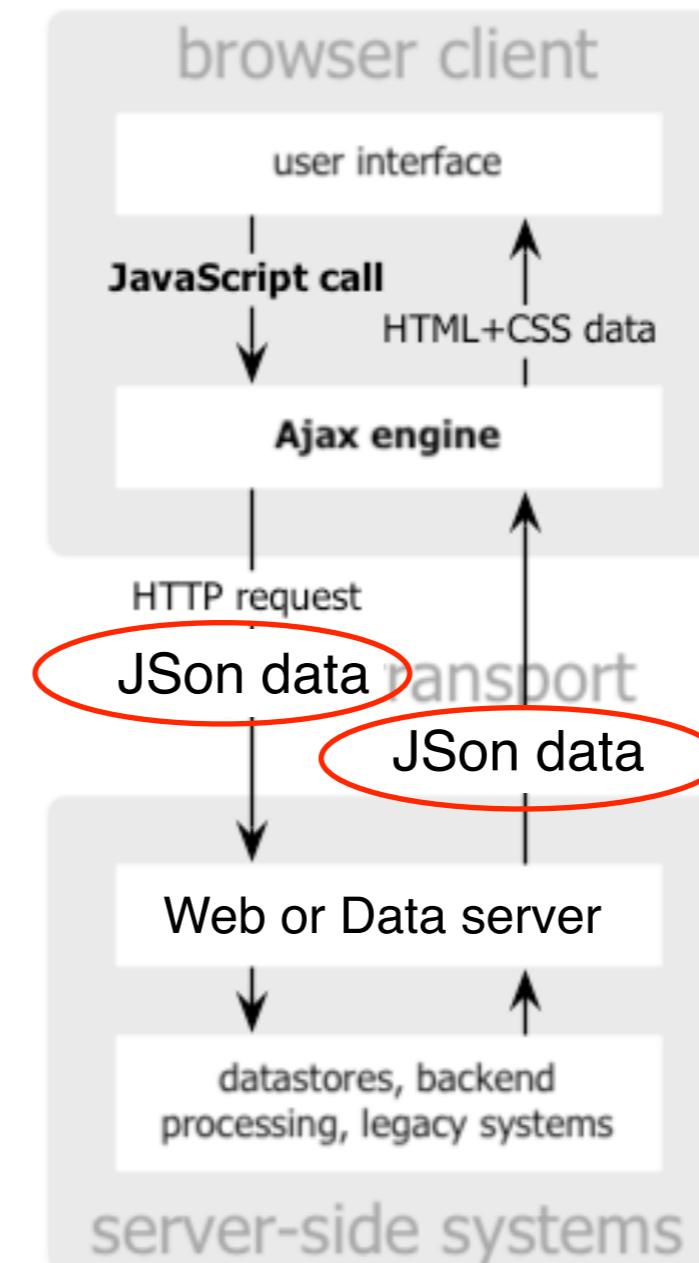
Ajax: A New Approach to Web Applications
by [Jesse James Garrett](#)
<http://adaptivepath.com/publications/essays/archives/000385>

- 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
- At the start of the session, the browser loads an Ajax engine (not just a Web page)
 - Written in JavaScript
 - This engine is responsible for both
 - rendering the user interface and
 - communicating 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.
- So the user is never staring at a blank browser window and an hourglass icon, waiting around for the server to do something.



classic
web application model

Ajax: A New Approach to Web Applications
by [Jesse James Garrett](#)
<http://adaptivepath.com/publications/essays/archives/000385.php>



Ajax
web application model

Ajax can exchange several data formats
we will use json



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

A little Ajax program

- Ajax uses a programming model with display and events.
These events are user actions, they call functions associated to elements of the web page.
- Interactivity is achieved with forms and buttons.
- The original AJAX way of programming is very complicated
 - we will use JQuery which makes interaction very simple



The original Ajax code

Create object

```
<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>
```

Post to server

What to do when
server responds

(processChange is a function that will be
called)

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

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

...
<script>
sendAjaxQuery(url, data){
    $.ajax({
        url: url,
        type: "POST",
        data: data,
        context: this,
        contentType: 'application/json',      the url to contact
        error: function () {                 declare the action (POST)
            // do something here
        },
        success: function (response) {       the data to send (no need to stringify if declared
            // do something here                      datatype is JSON - done automatically by jQuery)
        }
    });
}</script>
```

declare a Json interaction

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

http response code 200<=x<300



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

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

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

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

FORM

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Ajax form</title>
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js">
    </script>
</head>
<body>
<h1>This is my form</h1>
```

This is my form

First name:

Last name:

Send Data

2 /

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()));  
}  
  
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

```
<script>                                data must be JSON stringify-ed before calling aj
                                               
    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: ''
// 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)); You MUST declare type JSON
}); You MUST declare type JSON
and Stringify before returning

module.exports = router;
```

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



A value that
makes no sense

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**



The
University
Of
Sheffield.

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

The screenshot shows the 'Tools for Web Developers' section of the Google Developers website. The 'JavaScript' menu item is highlighted. A 'Get Started' button is prominent, with a tooltip 'Pause Your Code With Breakpoints'. Other sections include CSS, Console, Network, Storage, Command Menu, Mobile Simulation, DOM, and Snippets. The main content area features a tutorial titled 'Get Started with Debugging JavaScript in Chrome DevTools' by Kayce Basques, with a brief description and a video link.

Because of course you use a debugger when programming

If you do not, better learning 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

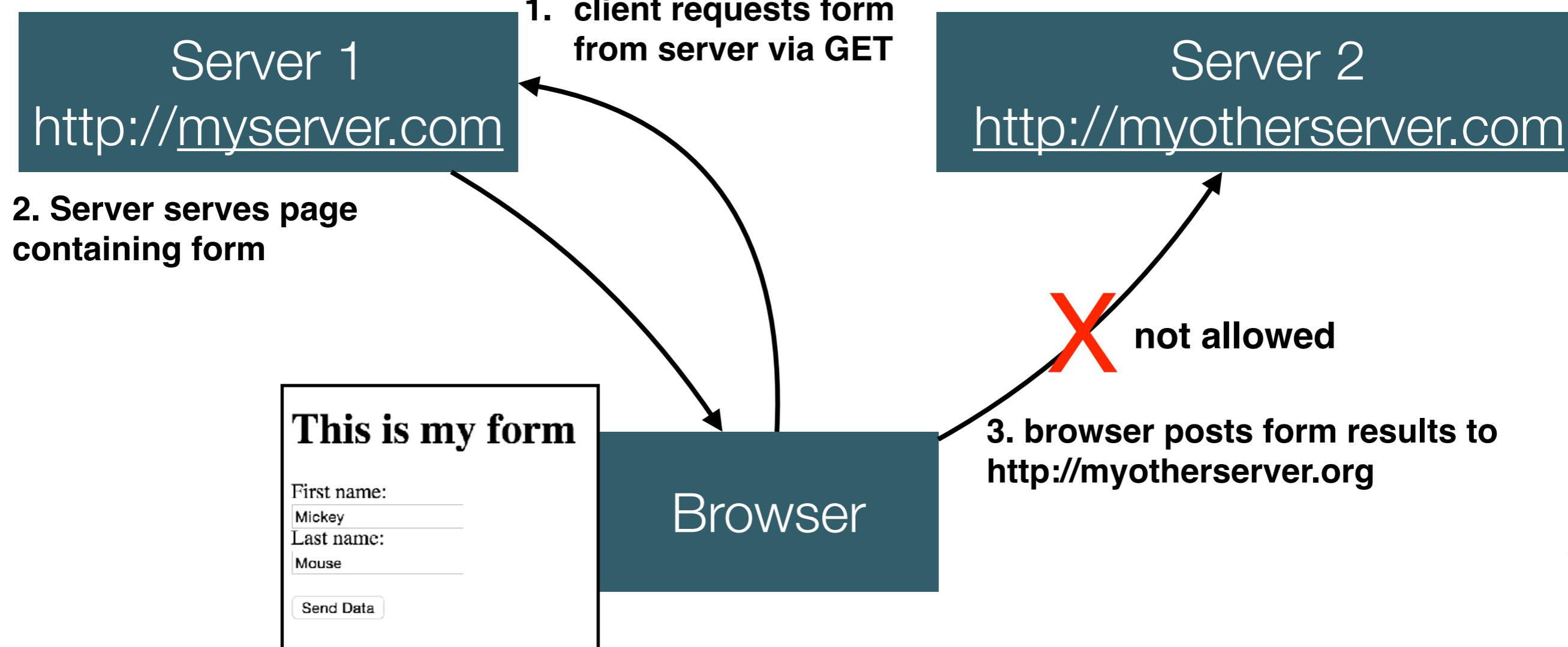


The
University
Of
Sheffield.

CORS



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://myotherserver.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 to the same server on another port. Origin '**http://myserver.org:63342**' is therefore not even allowed posting to '**http://myserver.org:3000**'.

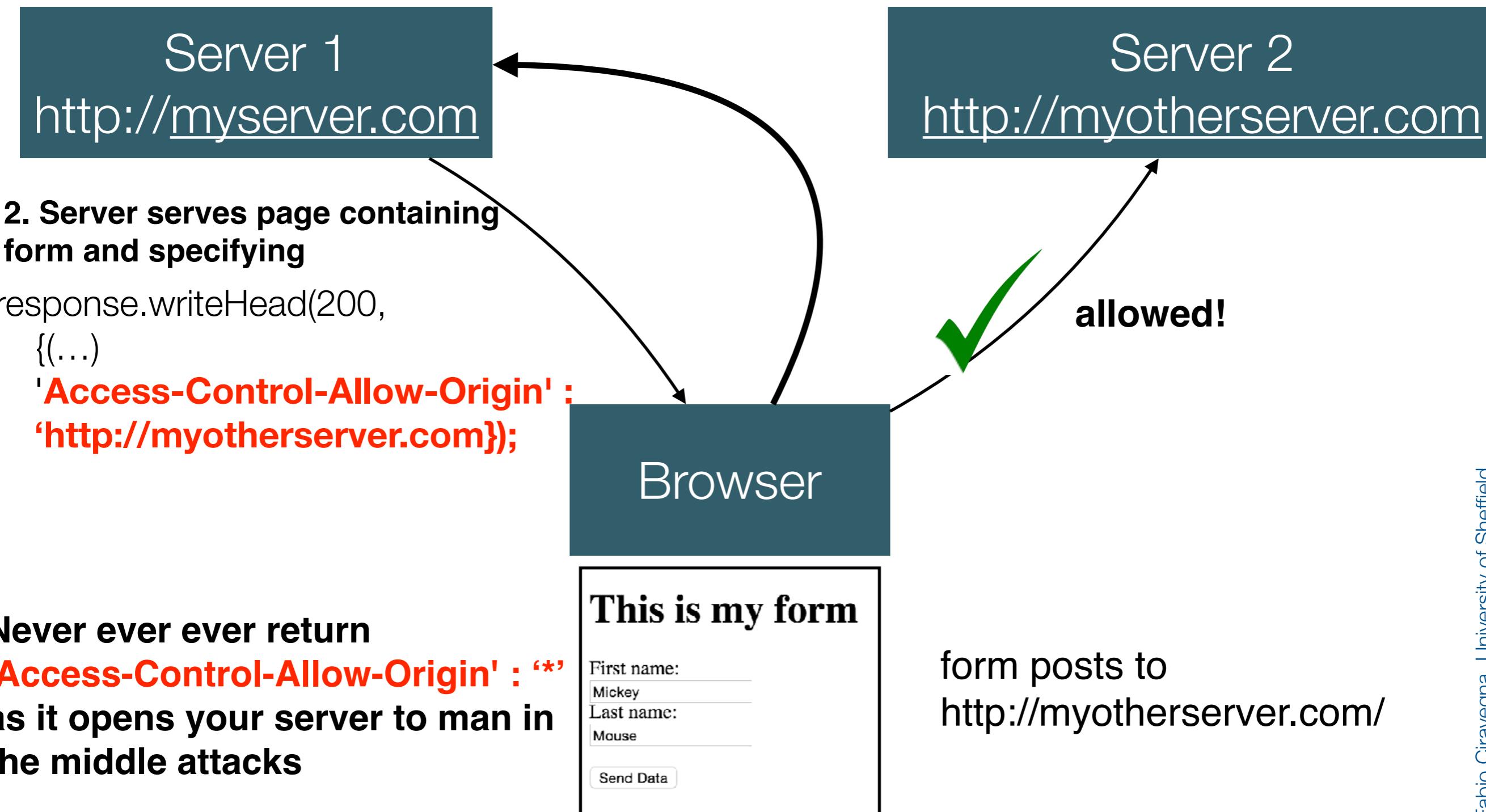
Beware!

- 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



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.js">
  </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.

More Express

Things you should be sure to know before starting doing something sophisticated in Express



The
University
Of
Sheffield.

Defining user variables

either lasting until you turn off the server
or lasting for the duration of the request

app.locals

app.locals

The `app.locals` object is a JavaScript object, and its properties are local variables within the application.

```
app.locals.title  
// => 'My App'
```

```
app.locals.email  
// => 'me@myapp.com'
```

Once set, the value of `app.locals` properties persist throughout the life of the application, in contrast with `res.locals` properties that are valid only for the lifetime of the request.

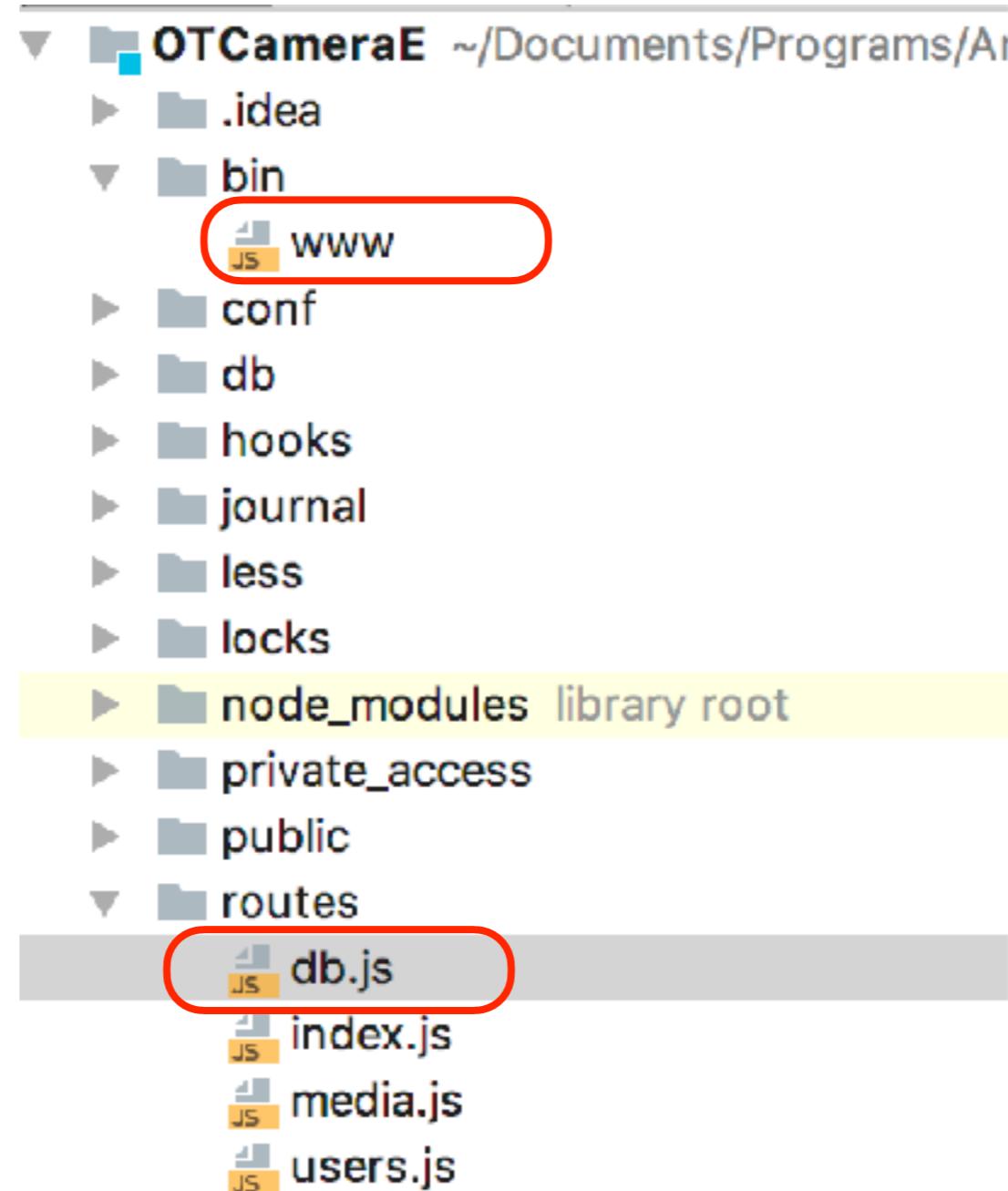
You can accesss local variables in templates rendered within the application. This is useful for providing helper functions to templates, as well as app-level data. Note, however, that you cannot access local variables in middleware.

```
app.locals.title = 'My App';  
app.locals.strftime = require('strftime');  
app.locals.email = 'me@myapp.com';
```

- **app.locals** variables persist for the lifetime of the server
 - so they are present for all user requests
- **res.locals** for the lifetime of the request
 - so the value is valid only for the specific user request — remember http is memoryless

why you want to use it?

- `app.locals` is typically used to pass data across routes and from the app to the routes





db route

```
router.post('/find_clients_from_remote', function (req, res) {
  console.log('find_clients_from_remote: ');
  var body = req.body;
  Database.getClient(body, function (err, dataX) {
    if (err || dataX.length == 0) {
      res.writeHead(400);
      res.end(err || 'user not found');
    } else {
      dataX = dataX[0];
      console.log('found client ' + JSON.stringify(dataX));
      var cidX = dataX._id.toString();
      console.log('cidx ' + cidX);

      // if cid is nonexistent it means it is not available
      if (isUndefined(cidX)) {
        console.log('cidx (NOT passed test) ' + cidX);
        res.writeHead(400);
        err = {status: 400, statusText: 'user not found'};
        res.end(JSON.stringify(err));
        return;
      }

      console.log('cidx (Passed test) ' + cidX);
      var npo = req.app.locals.getNewRoom();
      console.log(JSON.stringify(npo));
      var data = [
        cid: cidX,
```

ignore this part, it is irrelevant

app.locals is retrieved using the request object



```
function getNewRoom() {
  var ctrlRoomId = Math.floor(Math.random() * 10000) + 1;
  var useridCitizen = Math.floor((Math.random() * 10000) + 1);
  var roomid = Math.floor((Math.random() * 100000) + 1);
  var password = Math.floor((Math.random() * 100000) + 1);
  app.locals.allowedCtrlRooms ["CR" + ctrlRoomId] = "R" + roomid;
  app.locals.allowedCitRooms ["CT" + useridCitizen] = "R" + roomid;
  app.locals.validPorts ["R" + roomid] = "P" + password;
  return ([roomid, ctrlRoomId, useridCitizen, password]);
}
```

ignore this part, it is irrelevant

app.locals.getNewRoom= getNewRoom;

note: it is not assigned using the request object



Request parameter

<http://expressjs.com/4x/api.html>

req.hostname

Contains the hostname from the "Host" HTTP header.

```
// Host: "example.com:3000"  
req.hostname  
// => "example.com"
```

req.ip

The remote IP address of the request.

If the `trust proxy` is setting enabled, it is the upstream address; see [Express behind proxies](#) for more information.

```
req.ip  
// => "127.0.0.1"
```

req.path

Contains the path part of the request URL.

```
// example.com/users?sort=desc
req.path
// => "/users"
```

req.protocol

The request protocol string, "http" or "https" when requested with TLS. When the "trust proxy" [setting](#) trusts the socket address, the value of the "X-Forwarded-Proto" header ("http" or "https") field will be trusted and used if present.

```
req.protocol
// => "http"
```

req.query

An object containing a property for each query string parameter in the route. If there is no query string, it is the empty object, {}.

```
// GET /search?q=tobi+ferret
req.query.q
// => "tobi ferret"

// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse
req.query.order
// => "desc"
```

req.accepts(types)

Checks if the specified content types are acceptable, based on the request's `Accept` HTTP header field. The method returns the best match, or if none of the specified content types is acceptable, returns `undefined` (in which case, the application should respond with 406 "Not Acceptable").

The `type` value may be a single MIME type string (such as "application/json"), an extension name such as "json", a comma-delimited list, or an array. For a list or array, the method returns the **best** match (if any).

```
// Accept: text/html
req.accepts('html');
// => "html"

// Accept: text/*, application/json
req.accepts('html');
// => "html"
req.accepts('text/html');
// => "text/html"
req.accepts('json, text');
// => "json"
req.accepts('application/json');
// => "application/json"

// Accept: text/*, application/json
req.accepts('image/png');
req.accepts('png');
// => undefined

// Accept: text/*;q=.5, application/json
req.accepts(['html', 'json']);
req.accepts('html, json');
// => "json"
```

req.get(field)

Returns the specified HTTP request header field (case-insensitive match). The `Referrer` and `Referer` fields are interchangeable.

```
req.get('Content-Type');
```

```
// => "text/plain"
```

```
req.get('content-type');
```

```
// => "text/plain"
```

```
req.get('Something');
```

```
// => undefined
```

Aliased as `req.header(field)`.

req.is(type)

Returns `true` if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the `type` parameter. Returns `false` otherwise.

```
// With Content-Type: text/html; charset=utf-8
```

```
req.is('html');
```

```
req.is('text/html');
```

```
req.is('text/*');
```

```
// => true
```

```
// When Content-Type is application/json
```

```
req.is('json');
```

```
req.is('application/json');
```

```
req.is('application/*');
```

```
// => true
```



Getting parameters (POST)

<http://stackoverflow.com/questions/5710358/how-to-get-post-query-in-express-node-js>

```
var bodyParser = require('body-parser')
// to support JSON-encoded bodies
app.use(bodyParser.json());

//OR to support URL-encoded bodies (i.e. forms)
app.use(bodyParser.urlencoded({
  extended: true
}));

.. in your route

// assuming POST: name=foo&color=red      <-- URL encoding
// OR POST: {"name": "foo", "color": "red"}    <-- JSON encoding

app.post('/test-page', function(req, res) {
  var name = req.body.name,
      color = req.body.color;
}) ;
```



Response object

res.attachment([filename])

Sets the HTTP response `Content-Disposition` header field to "attachment". If a `filename` is given, then it sets the `Content-Type` based on the extension name via `res.type()`, and sets the `Content-Disposition` "filename=" parameter.

```
res.attachment();
// Content-Disposition: attachment

res.attachment('path/to/logo.png');
// Content-Disposition: attachment; filename="logo.png"
// Content-Type: image/png
```

res.end([data] [, encoding])

Ends the response process. Inherited from Node's [http.ServerResponse](#).

Use to quickly end the response without any data. If you need to respond with data, instead use methods such as `res.send()` and `res.json()`.

```
res.end();
res.status(404).end();
```

res.json([body])

Sends a JSON response. This method is identical to `res.send()` with an object or array as the parameter. However, you can use it to convert other values to JSON, such as `null`, and `undefined`. (although these are technically not valid JSON).

```
res.json(null)
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```



Response methods

The methods on the response object (`res`) in the following table can send a response to the client and terminate the request response cycle. If none of them is called from a route handler, the client request will be left hanging.

Method	Description
<code>res.download()</code>	Prompt a file to be downloaded.
<code>res.end()</code>	End the response process.
<code>res.json()</code>	Send a JSON response.
<code>res.jsonp()</code>	Send a JSON response with JSONP support.
<code>res.redirect()</code>	Redirect a request.
<code>res.render()</code>	Render a view template.
<code>res.send()</code>	Send a response of various types.
<code>res.sendFile</code>	Send a file as an octet stream.
<code>res.sendStatus()</code>	Set the response status code and send its string representation as the response body.

By the way Jsonp

<https://en.wikipedia.org/wiki/JSONP>

- JSONP is a trick that allows a web browser
 - to fetch JSON data from a JSON server and
 - feed them to a Javascript script in the browser.

```
<script type="application/javascript"
       src="http://server.example.com/Users/1234?callback=parseResponse">
</script>
...
<script>
  function parseResponse(param) {
    ...
  }
</script>
```



Response: redirection

res.redirect([status,] path)

Express passes the specified URL string as-is to the browser in the `Location` header, without any validation or manipulation, except in case of `back`.

Browsers take the responsibility of deriving the intended URL from the current URL or the referring URL, and the URL specified in the `Location` header; and redirect the user accordingly.

Redirects to the URL derived from the specified path, with specified [HTTP status code](#) status. If you don't specify status, the status code defaults to "302 "Found".

```
res.redirect('/foo/bar');
res.redirect('http://example.com');
res.redirect(301, 'http://example.com');
res.redirect('../login');
```

Redirects can be a fully-qualified URL for redirecting to a different site:

```
res.redirect('http://google.com');
```

Redirects can be relative to the root of the host name. For example, if the application is on `http://example.com/admin/post/new`, the following would redirect to the URL `http://example.com/admin`:

```
res.redirect('/admin');
```

A `back` redirection redirects the request back to the `referer`, defaulting to / when the referer is missing.

```
res.redirect('back');
```

note! useful for login requests!!!

res.send([body])

Sends the HTTP response.

The `body` parameter can be a `Buffer` object, a `String`, an object, or an `Array`. For example:

```
res.send(new Buffer('whoop'));
res.send({ some: 'json' });
res.send('<p>some html</p>');
res.status(404).send('Sorry, we cannot find that!');
res.status(500).send({ error: 'something blew up' });
```

This method performs many useful tasks for simple non-streaming responses: For example, it automatically assigns the `Content-Length` HTTP response header field (unless previously defined) and provides automatic HEAD and HTTP cache freshness support.

When the parameter is a `Buffer` object, the method sets the `Content-Type` response header field to "application/octet-stream", unless previously defined as shown below:

```
res.set('Content-Type', 'text/html');
res.send(new Buffer('<p>some html</p>'));
```

When the parameter is a `String`, the method sets the `Content-Type` to "text/html":

```
res.send('<p>some html</p>');
```

When the parameter is an `Array` or `Object`, Express responds with the JSON representation:

```
res.send({ user: 'tobi' });
res.send([1,2,3]);
```



res.sendStatus(statusCode)

Set the response HTTP status code to `statusCode` and send its string representation as the response body.

```
res.sendStatus(200); // equivalent to res.status(200).send('OK')
res.sendStatus(403); // equivalent to res.status(403).send('Forbidden')
res.sendStatus(404); // equivalent to res.status(404).send('Not Found')
res.sendStatus(500); // equivalent to res.status(500).send('Internal Server Error')
```

If an unsupported status code is specified, the HTTP status is still set to `statusCode` and the string version of the code is sent as the response body.

```
res.sendStatus(2000); // equivalent to res.status(2000).send('2000')
```

res.set(field [, value])

Sets the response's HTTP header `field` to `value`. To set multiple fields at once, pass an object as the parameter.

```
res.set('Content-Type', 'text/plain');

res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  'ETag': '12345'
})
```

Aliased as `res.header(field [, value])`.

res.status(code)

Use this method to set the HTTP status for the response. It is a chainable alias of Node's [response.statusCode](#).

```
res.status(403).end();
res.status(400).send('Bad Request');
res.status(404).sendFile('/absolute/path/to/404.png');
```

res.type(type)

Sets the `Content-Type` HTTP header to the MIME type as determined by [mime.lookup\(\)](#) for the specified `type`. If `type` contains the "/" character, then it sets the `Content-Type` to `type`.

```
res.type('.html');           // => 'text/html'
res.type('html');            // => 'text/html'
res.type('json');            // => 'application/json'
res.type('application/json'); // => 'application/json'
res.type('png');             // => image/png:
```



IntelliJ does it for you

Express application generator

Use the application generator tool, `express`, to quickly create a application skeleton.

Install it with the following command.

```
$ npm install express-generator -g
```

Display the command options with the `-h` option:

```
$ express -h
```

```
Usage: express [options] [dir]
```

```
Options:
```

<code>-h, --help</code>	output usage information
<code>-V, --version</code>	output the version number
<code>-e, --ejs</code>	add ejs engine support (defaults to jade)
<code>--hbs</code>	add handlebars engine support
<code>-H, --hogan</code>	add hogan.js engine support
<code>-c, --css <engine></code>	add stylesheet <engine> support (less stylus compass) (defaults to plain css)
<code>-f, --force</code>	force on non-empty directory



For example, the following creates an Express app named **myapp** in the current working directory.

```
$ express myapp

create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/public
create : myapp/public/javascripts
create : myapp/public/images
create : myapp/routes
create : myapp/routes/index.js
create : myapp/routes/users.js
create : myapp/public/stylesheets
create : myapp/public/stylesheets/style.css
create : myapp/views
create : myapp/views/index.jade
create : myapp/views/layout.jade
create : myapp/views/error.jade
create : myapp/bin
create : myapp/bin/www
```

Then install dependencies:

```
$ cd myapp
$ npm install
```



The
University
Of
Sheffield.

Connecting to MySQL

Just in case you ever need it

Module: [mysql](#)
Installation

<http://expressjs.com/guide/database-integration.html>

```
$ npm install mysql
```

Example

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'dbuser',
  password  : 's3kreee7'
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function(err, rows, fields) {
  if (err) throw err;
  console.log('The solution is: ', rows[0].solution);
});

connection.end();
```

The callback has the same syntax
as the standard nodejs one



The
University
Of
Sheffield.

Questions?
