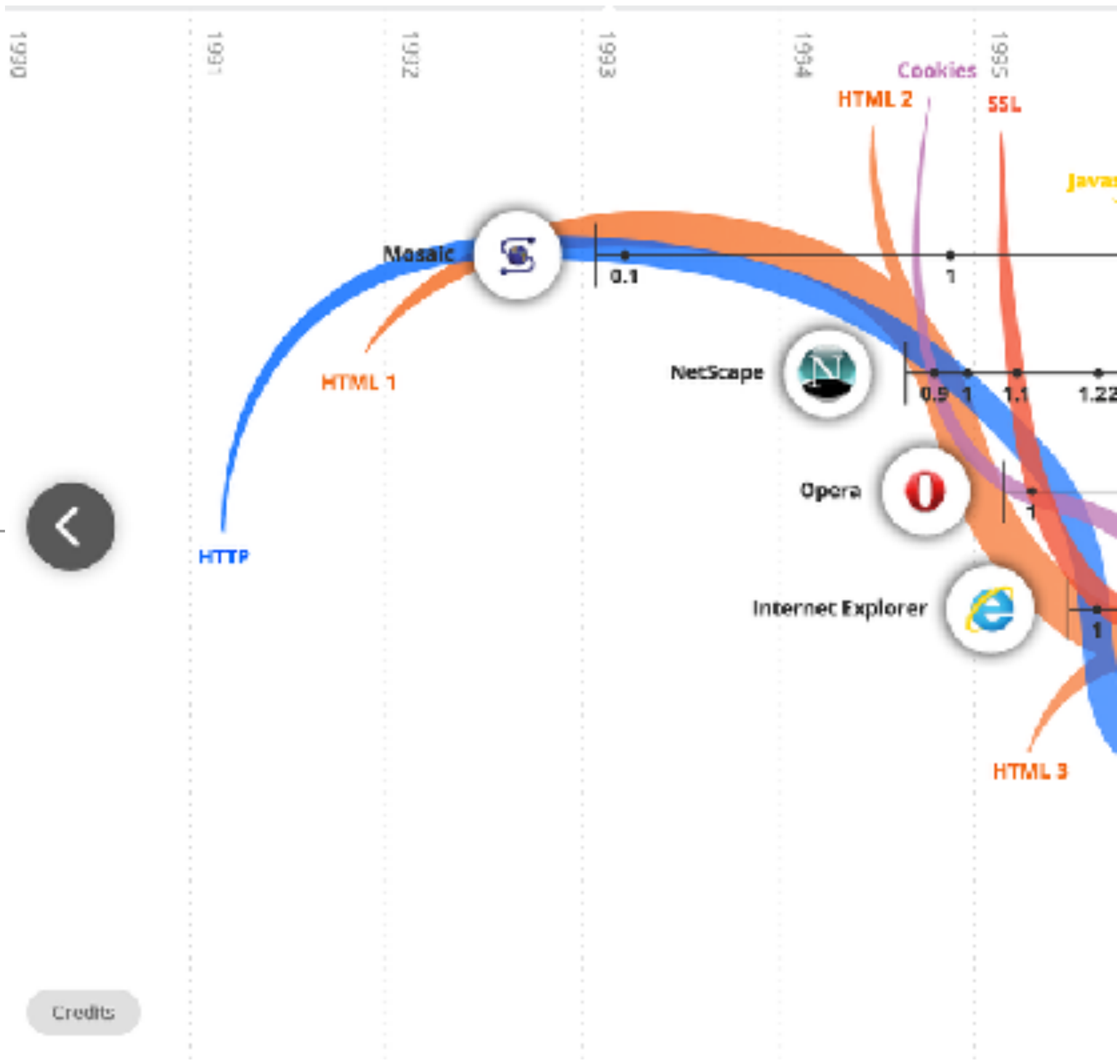
Accessing the Web - Client server architecture with Node.js

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

The University Of Sheffield.



The Http Protocol

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

we will see request and response as parameter of any callback to the server in NodeJS

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

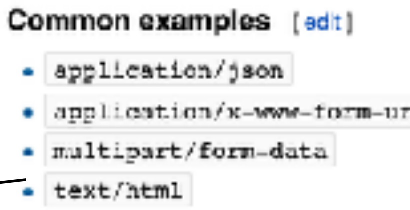
HTTP: GET and POST methods

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

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

Request Headers

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



Common examples [edit]

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

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

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)

The University Of Sheffield.

Please note!

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

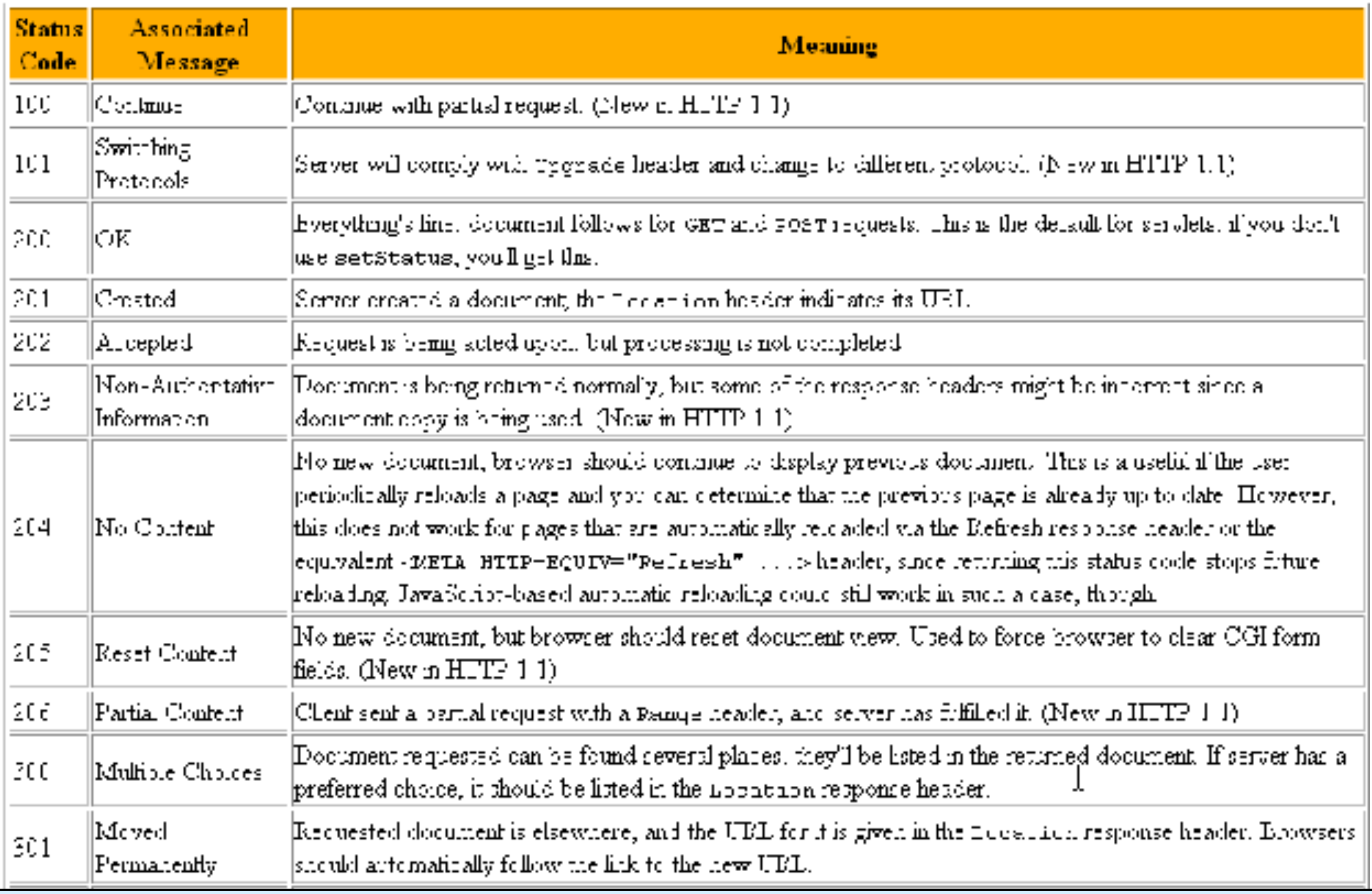
Please note very often you will use the http request directly (e.g. via a direct call). In other cases you will use some forms of Web API (e.g. Twitter API). However all these APIs have equivalent information that is either set up automatically (e.g. user agent in the Twitter API) or via the parameters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

Not setting these means low marks in the assignment!

Responses Codes

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

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



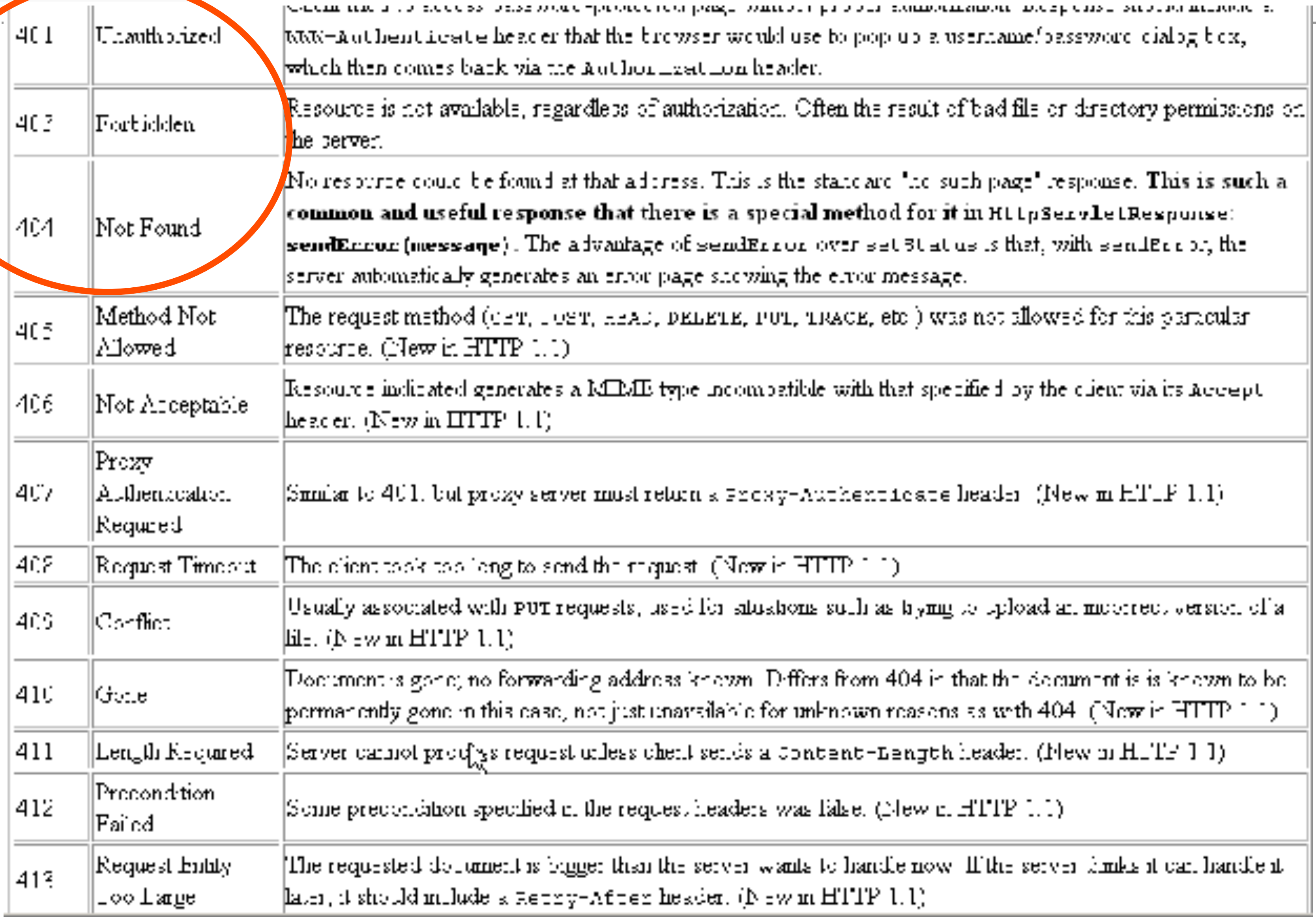
Status Code	Associated Message	Meaning
100	Continue	Continue with partial request. (New in HTTP 1.1)
101	Switching Protocols	Server will comply with Upgrade header and change to different protocol. (New in HTTP 1.1)
200	OK	Everything's fine; document follows for GET and POST requests. This is the default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

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

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

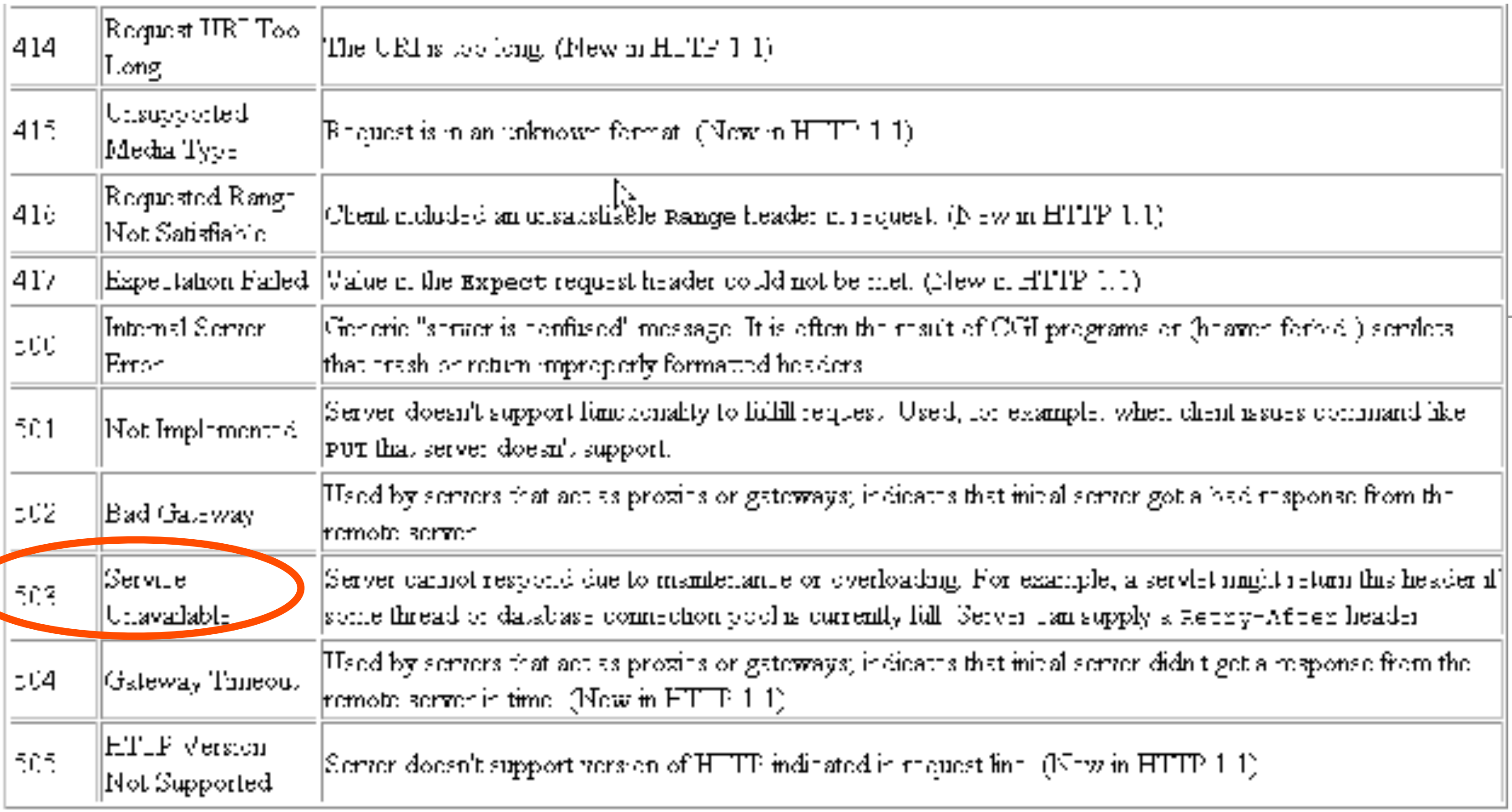
The University Of Sheffield.

Responses (2)



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

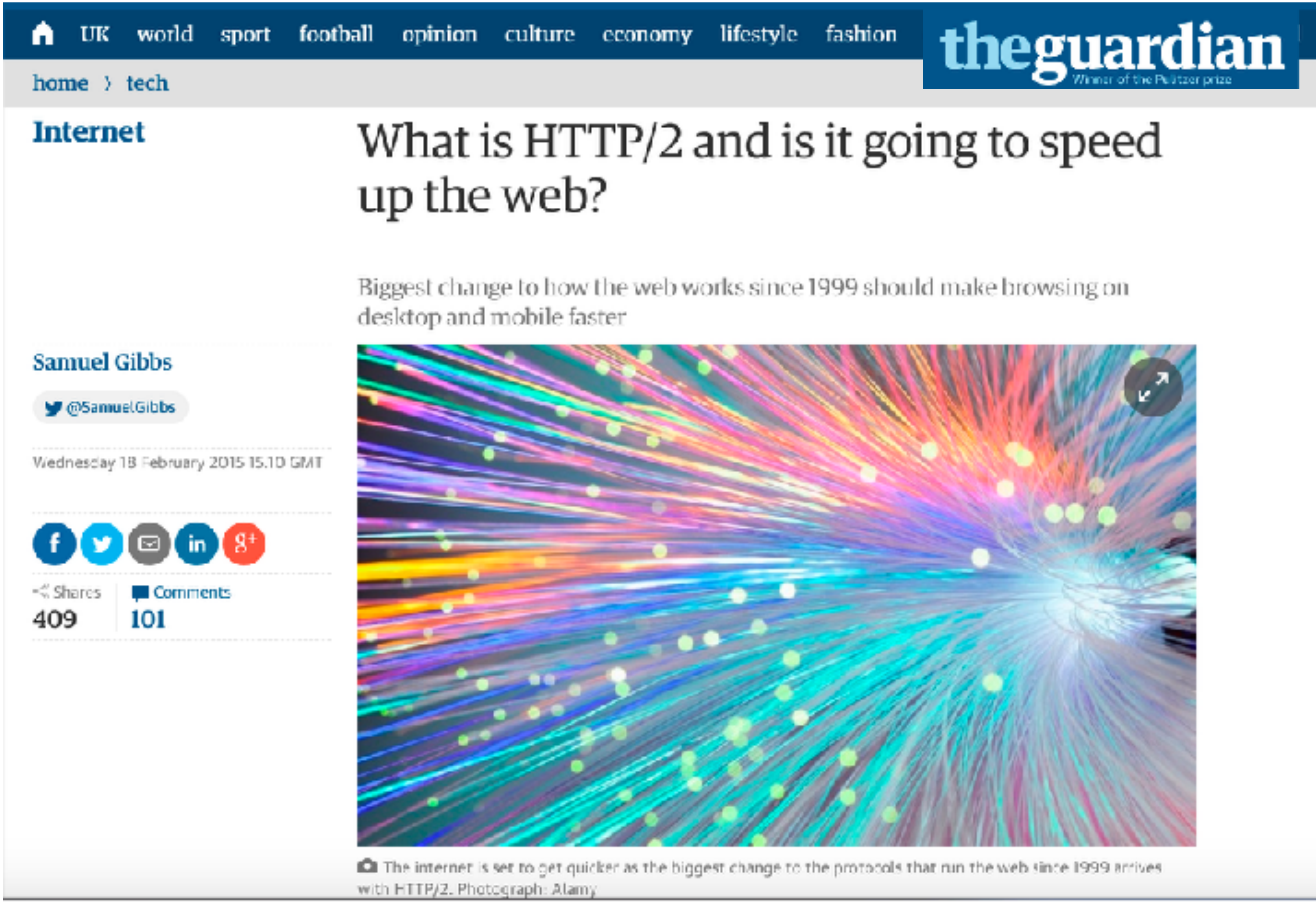
http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/



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!!!

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


UK world sport football opinion culture economy lifestyle fashion
theguardian
Winner of the Pulitzer prize
home › tech
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

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 minimize 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.

Building a simple Client Server Architecture using HTML and Javascript

http://nodejs.org/

The University Of Sheffield.

Client Server Architecture

nodejs.org/

- In a typical client server architecture a client (e.g. a browser) communicates with a server (e.g. an Apache Tomcat server a Node server) to obtain data (e.g. a web page)
- You will have seen HTML and Javascript as a way to provide means to ask (GET) and send (POST) information to a server
- Building a server has been historically complex
- Node.js enables writing the server using Javascript
  - And it is very simple to use
  - There are a huge numbers of modules (libraries) that can installed to add features and functionality – like data stores, Zip file support, Facebook login, or payment gateways.

Node.js

nodejs.org/

- Node.js is a platform built on Chrome's JavaScript runtime V8 for easily building fast, scalable network applications
- Node.js uses a model that makes it lightweight and efficient,
  - perfect for data-intensive real-time applications that run across distributed devices
- Model:
  - Event-driven
  - Non-blocking I/O

Node.Js

http://www.makeuseof.com/tag/what-is-node-js-and-why-should-i-care-web-development/

- Event based:
  - JavaScript is an event-based language, so anything that happens on the server triggers a non-blocking event
    - Each new connection fires an event; data being received from an upload form fires a data-received event;
    - requesting data from the database fires an event.
  - In practice, this means a Node site will never lock up and can support tens of thousands of concurrent users.

Why Use Node.js?

http://www.makeuseof.com/tag/what-is-node-js-and-why-should-i-care-web-development/

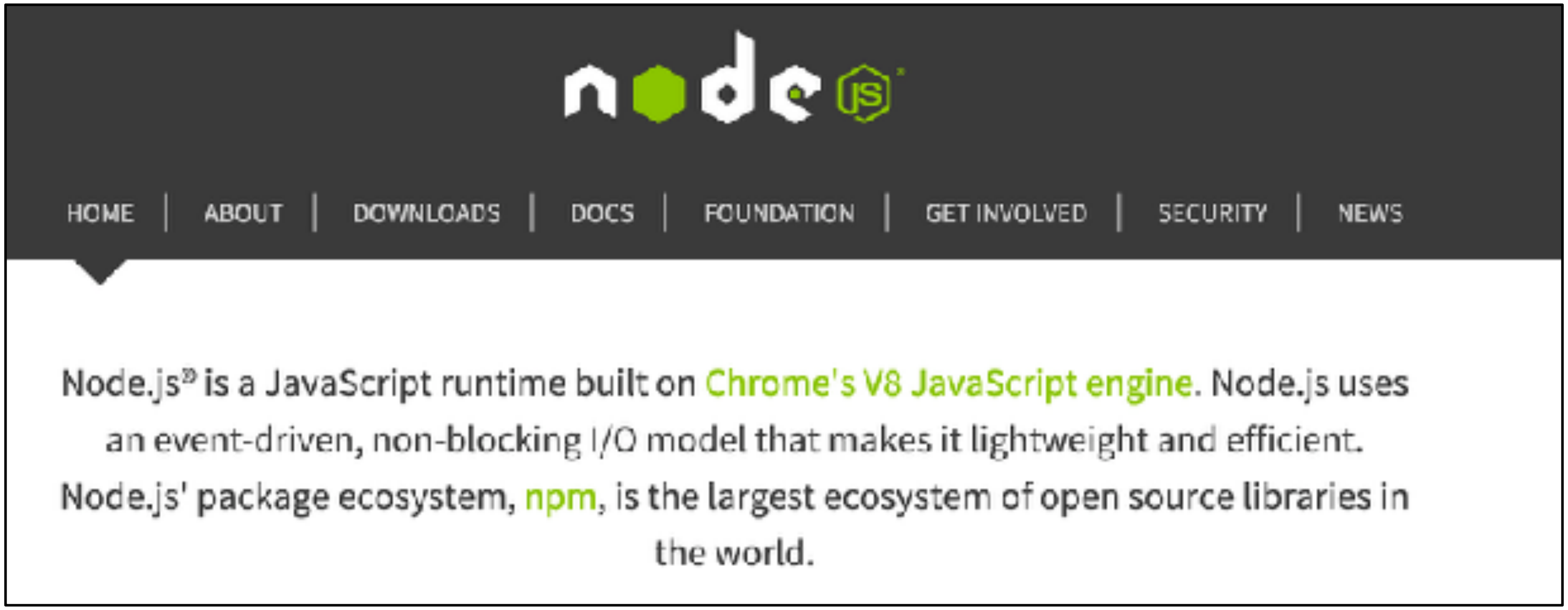
RESTful systems typically communicate over Hypertext Transfer Protocol (HTTP) with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) that web browsers use to retrieve web pages and to send data to remote servers.
REST systems interface with external systems as web resources identified by Uniform Resource Identifiers (URIs), for example /people/tom, which can be operated upon using standard verbs such as DELETE /people/tom.
https://en.wikipedia.org/wiki/Representational_state_transfer

- Performance and sc
    - Node is fast
- Node is perfect for c
    - A web service which takes a few input parameters and passes a little data back
        - Simple data manipulation without a huge amount of computation.
    - Node can handle thousands of these concurrently where PHP would just collapse.
- It is Javascript: simple and powerful

The University Of Sheffield.

Documentation

http://nodejs.org/api/



Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Why Node.js

http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

- In one sentence: Node.js shines in real-time web applications employing push technology over websockets.
- What is so revolutionary about that?
  - Well, after over 20 years of stateless-web based on the stateless request-response paradigm,
  - we finally have web applications with real-time, two-way connections,
    - where both the client and server can initiate communication,
    - allowing them to exchange data freely. This is in stark contrast to the typical web response paradigm, where the client always initiates communication

Drawbacks

http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

- You definitely don't want to use Node.js for CPU-intensive operations;
  - Using it for heavy computation will annul nearly all of its advantages
  - Node really shines in building fast, scalable network applications,
    - as it's capable of handling a huge number of simultaneous connections with high throughput,
    - which equates to high scalability.

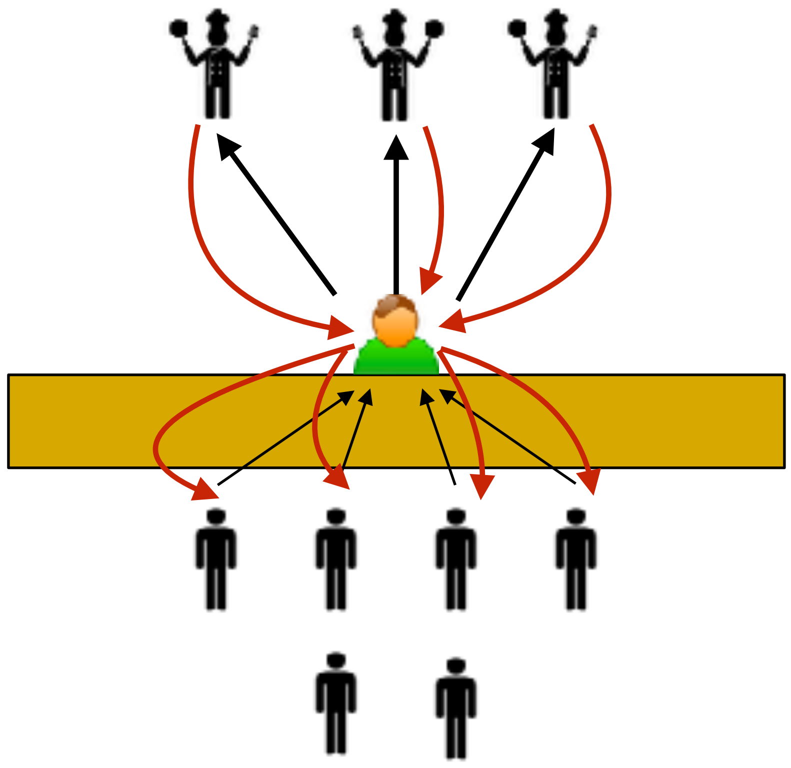
Why?

http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

- Node.js is single threaded
  - i.e. there is just one server at the till of the fast-food outlet
  - Heavy computation could choke up Node's single thread and cause problems for all clients
    - As incoming requests would be blocked until said computation was completed.
    - In a fast food outlet the cashier also manages the fries (and when it does it blocks the queue - imagine if he also cooked the burgers and cleaned the floor!)
- Moreover, an exception bubbling up to the core (topmost) Node.js event loop,
  - will cause the Node.js instance to terminate
    - effectively crashing the server!

Use non-blocking events

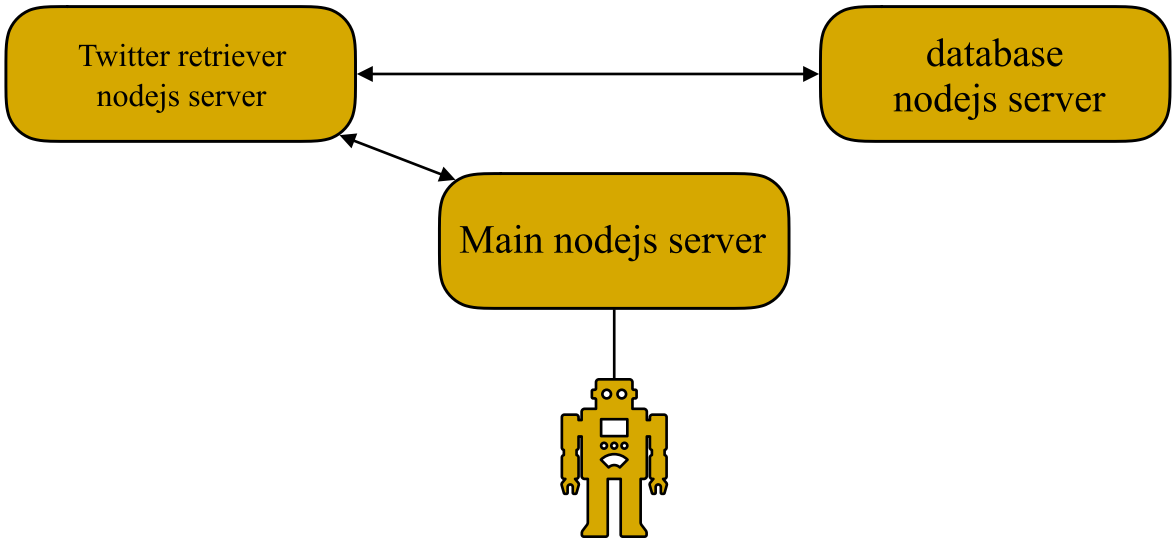
- Organise a node server as a burger joint



- Requests are posted to the till (the node.js server) which will direct them to the right cook (e.g. a database)
  - the till is not blocked by the time needed to prepare the food
  - While the food is being prepared, the till can serve other requests
- When the cook has prepared the food (the data), the counter (node.js server) will return it to the client

NodeJs as a burger joint

- So organise your server so that the main loop (capturing the http request event) is never blocked by heart computations
- Use a small constellation of fast specialised nodejs servers around it doing the computation



NPM: node.js packages

http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

- Package management is supported using the NPM tool that comes by default with every Node.js installation.
- NPM modules are similar to Ruby Gems:
  - a set of publicly available, reusable components, available through easy installation
    - via an online repository,
    - with version and dependency management
- A full list of packaged modules can be found on the NPM website https://npmjs.org/

To Install an NPM package

https://docs.npmjs.com/cli/install

- global installation
  - installation as a global package; visible by all applications using node.js
  - **npm install <package name or url> -g**
  - This is unlikely to work on the lab computers!
    - You may not have permission to do so
- application installation
  - **npm install <package name or url>**
  - will only be recognised for the current application
    - do this on lab computers
  - this will create a **node_modules** folder under your app folder containing all the node packages installed

Well Known NPM Modules

http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

- **express** - Express.js, a web development framework for Node.js, and the de-facto standard for the majority of Node.js applications out there today.
- **connect** - Connect is an extensible HTTP server framework for Node.js, providing a collection of high performance “plugins” known as middleware; serves as a base foundation for Express.
- **socket.io** and sockjs - Server-side component of the two most common websockets components out there today.
- **Jade** - One of the popular templating engines, a default in Express.js.
- **mongo** and mongojs - MongoDB wrappers to provide the API for MongoDB object databases in Node.js.
- **passport -** for authentication

Examples of node.js usage

http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

- Chat is the most typical real-time, multi-user application.
  - Node.js with websockets running over the standard port 80.
  - We will see it in Lecture 6
- Api on top of a networked Database
  - Especially if lots of input is provided
    - Non-blocking operations cope with that
- Data streaming
  - We will see it in Lecture 6 (videoconference system using WebRTC)

Our first example

- In the lab we will use IntelliJ, a development environment similar to Eclipse
    - but easy to use and very powerful
- IntelliJ will build the backbone of our server when we create a default Nodejs project
    - Using Express
- However it is useful see how a NodeJS program works without Express

The cycle is always the same

- Create a nodejs project in IntelliJ
- Run it using IntelliJ
- Open Chrome on the local host (unless you deploy to a cloud server)
  http://localhost:<selected port>/
  - e.g. https://localhost:3000/

A basic server

```
var http = require(‘http');

http.createServer(
  function (request, result) {
    result.writeHead(200, {'Content-Type': 'text/
  plain'});
    result.end('Hello World\n');
}).listen(3000);
```

A base server

http is a node module

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

- creates an instance of an object of type http

```
http.createServer(
  function (request, result) {
    result.writeHead(200, {'Content-Type': 'text/plain'});
    result.end('Hello World\n');
}).listen(3000);
```

- createServer is a method in the http module that creates an event listener for any connection

```
console.log('Server running at http://localhost:3000/');
```

- writes on the node.js command window

ctd

```
result.writeHead(200, {'Content-Type': 'text/plain'});
```

- this returns code 200 (ok) to the client

```
result.end('Hello World\n');
```

- it sends back the strings to the client and closes the communication

A basic server

When the server receives an https request
with request and response, nodejs will invoke this callback

the request is a Javascript object containing - among other things - the parameter passed by the client (e.g. the fields of a form).

```
http.createServer(
  function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/
  plain'});
    response.end(‘Hello World\n');
}).listen(3000);
```

The server must ALWAYS return an HTTP code by writing into the header of the response

whatever we write in the response, it will go back to the client.
this will be a JSON object (see next week).
If you have never seen JSON, think of it as a Javascript object

Asynchronous == Callbacks

```
http.createServer(
  function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  });
```

callback functions are a rather peculiar thing. It is a case where the code is a form of data, i.e. they can be passed as parameters. This is not allowed in imperative languages like Java but it used to be standard in languages like Lisp

Lisp is a family of computer programming languages based on formal functional calculus.
Lisp (for "List Processing Language") stores and manipulates programs in the same manner as any other data, making it well suited for "meta-programming" applications

https://en.wikiquote.org/wiki/Lisp_(programming_language)

(Lisp was the language of Artificial Intelligence in the 80s)

Parameters to the http.createServer callback

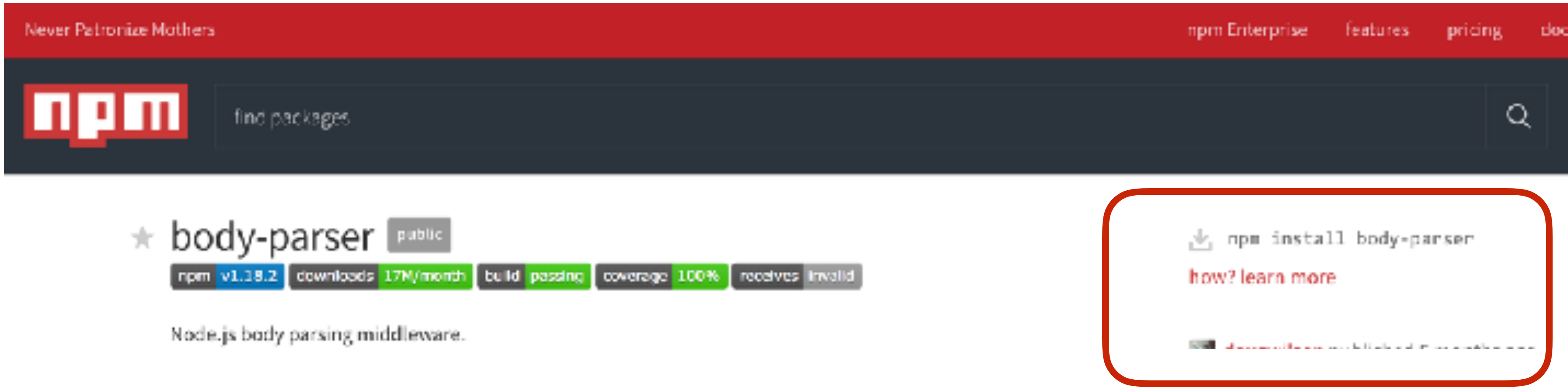
```
function (request, result) {
```

- request:
  - the request sent by the client inclusive of its parameters (if any)
  - e.g. $\underbrace{\text{http://}}_{\text{protocol}}\underbrace{\text{localhost}}_{\text{server}}:\underbrace{\text{3000}}_{\text{port}}/\underbrace{\text{index.html}}_{\text{file}}\underbrace{\textbf{?id=123\&room=2}}_{\text{parameters}}$

- result:
  - is a placeholder for the reply generated by the server
    - we will put here the values that the createServer returns (html code or data)

Nody-parser

- It enable accessing the parameters provided by the client


Never Patronize Mothers

npm Enterprise features pricing doc

find packages

body-parser public

npm v1.18.2 downloads 17M/month build passing coverage 100% receives invalid

Node.js body parsing middleware.

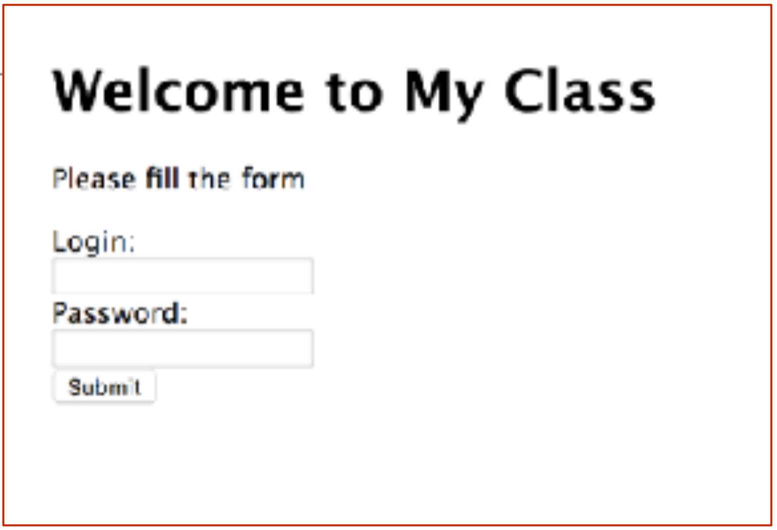
```
npm install body-parser
```

how? learn more

NPM modules always give you the command to use to install the module (use that on a command line)

How to use body-parser

To respond to a form (see today's lab)


Welcome to My Class

Please fill the form

Login:

Password:

```
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: false }));

…

router.post('/index', function(req, res, next) {
    var login= req.body.login;
    var password= req.body.password;

…
}
```

with body-parser the form fields become fields in the request **body**
In general, any data sent from the client will become field in the request

We will see what **router.post** is. For now just check how to access the parameters **login and password** from a form

The University Of Sheffield.

Getting the request headers

http://nodejs.org/api/http.html

- Getting the request headers (e.g. user-agent)

```
HTTP message headers are represented by an object like
this:

{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'host': 'mysite.com',
  'accept': '*/*' }
```

- a value is requested as
    - **request.headers**['user-agent']
- this enables to access the https protocol parameters (and hence e.g. to the request metadata - e.g. it comes from a Chrome browser)
- please note!
    - all fields are lowercase, values are not modified

GET request: Accessing the parameters

```
var http = require('http');
var url = require('url');

var server = http.createServer(function (request, response) {
  if (request.method == 'GET') {
    var queryData = req.body;
    response.writeHead(200, {"Content-Type": "text/plain"});
   // if parameter is provided
     if (queryData.name) {
       response.end('Hello ' + queryData.name + '\n');
      } else {
       response.end("Hello World\n");
     }
  }
});

// Listen on port 3000, IP defaults to 127.0.0.1 (localhost)
server.listen(3000);
```

POST

- A post sends (complex) data
  - which will arrive in chunks
- To get the request Node.js will do two things:
  - will receive the request
  - will set a listener for the event of receiving a chunk of data
    - request.on('data'
  - will set a listener for a signal that the data upload is completed
    - request.on('end'

Returning values or existing files

- With node.js you can either return
  - values or data via the response parameter

```
response.end(‘Hello world’);
(we will see Json next week)
```

  - and/or
    - existing files

```
var file = new (static.Server)();
file.serve(req, res);
```

Serving a file

```
var protocol = require('http');
var static = require('node-static');
var util = require('util');
var file = new (static.Server)();
var portNo = 3000;
var app = protocol.createServer( function (req, res) {
    file.serve(req, res, function (err, result) {
          if (err!=null) {
               console.error('Error serving %s - %s', req.url,
                    err.message);
               if (err.status === 404 || err.status === 500) {
                    file.serveFile(util.format('/%d.html', err.status),
                          err.status, {}, req, res);
               } else {
                    res.writeHead(err.status, err.headers);
                    res.end();
               }
          } else {
               console.log('serving %s (err: %s)', req.url, err);
          }
     });
}).listen(portNo);
```

this requires to have performed **npm install node-static**

please note you should check that the request is a get before file.serve (and also that you are requested a url)

Callback function in case of error

there is an error

serve the file 404.html (you must provide one!)

return the appropriate error status

file.serve will return the file. here we just tell the c

Responding to an end point

- Most times you will not have a physical file: for example you will generate one from database data (e.g. in e-commerce)
- Pathname requested:
    - var pathname = url.parse(request.url).pathname;
    - this will return (e.g.) /index.html
- Pathname is the path section of the URL, that comes
    - after the host
    - before the query
    - it includes the initial slash if present

Modifying our function

```
function (req, res) {
  var pathname = url.parse(req.url).pathname;
 if (pathname==‘/virtual_end_point.html’){
   res.writeHead(200, {"Content-Type": "text/plain"});
   res.end(return a value’);
   }
  else
    file.serve(req, res, function (err, result) {
       if (err!=null) {
           console.error('Error serving %s - %s', req.url,
              err.message);
           if (err.status === 404 || err.status === 500) {
               file.serveFile(util.format('/%d.html', err.status),
                    err.status, {}, req, res);
           } else {
               res.writeHead(err.status, err.headers);
               res.end();
           }
       } else {
           console.log('serving %s (err: %s)', req.url, err);
       }
   }
```

The University Of Sheffield.

Routing and Express

Routing

- Routing refers to determining how an application responds to a client request to a particular endpoint,
  - which is a URI (or path) and
  - a specific HTTP request method (GET, POST, and so on)
- Each route can have one or more handler functions
  - which is / are executed when the route is matched

Nodejs Get Routing to /

```
var http = require('http');
var url= require('url');
var server = http.createServer(function (request, response) {
    var pathname = url.parse(req.url).pathname;
    if ((pathname=='/')&& (request.method == 'GET')) {
    response.end('Hello World');
    }});
server.listen(3000);
```

The app starts a server and listens on port 3000 for connection.
It will respond with "Hello World!" for requests to the homepage.
For every other path, it will respond with a 404 Not Found.

Express

http://expressjs.com/

- Node.js is great but most of its functions are rather verbose
- Express
  - A minimal and flexible node.js web application framework with a robust set of features for web applications
  - With a myriad of HTTP utility methods and middleware at your disposal
    - Creating a robust API is quick and easy
  - A thin layer of fundamental web application features, without obscuring Node features that you know and love

In express

```
var express = require('express')
var app = express()
var server = app.listen(3000);

app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

Every time we receive a get for “/“, then send back a string called Hello World

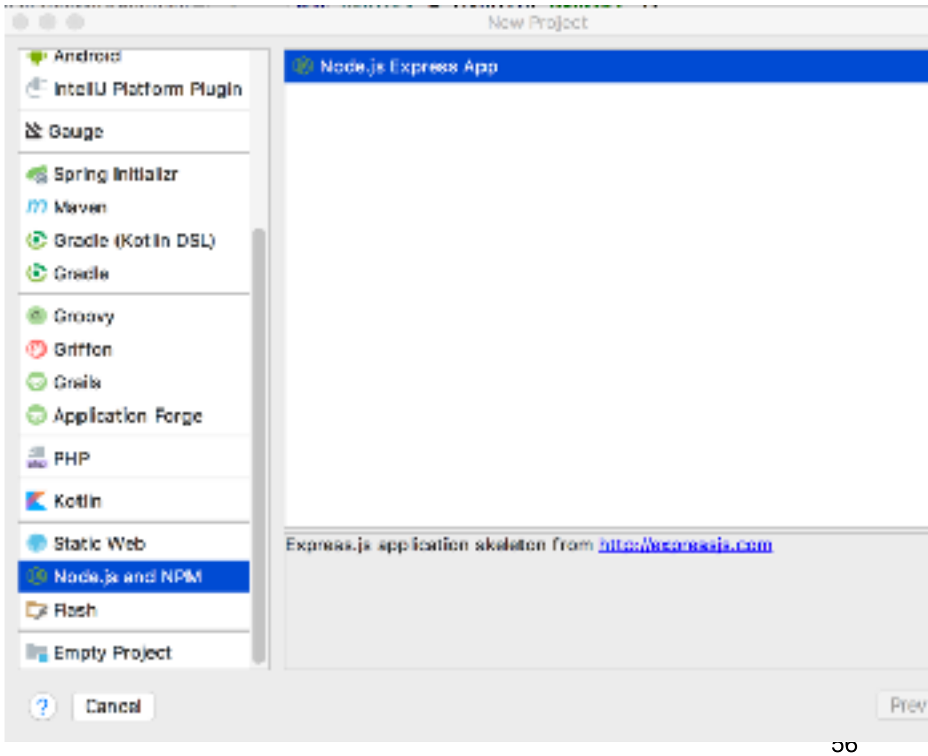
Routing in Express

- Route definition:
  - app is an instance of express
  - METHOD is an HTTP request method (POST, GET)
  - PATH is a path on the server,
  - HANDLER is the callback function executed when the route is matched

```
app.METHOD(PATH, CALLBACK)
```

How to create a project

- Intellij provides an excellent plugin for node projects
- It will use Express
- To create a project:
    - file>new>project
    - if you do not see node then you do not have the plugin -> install under Settings/Preference plugins



Express.js application skeleton from http://expressjs.com

Routing Examples

METHOD

```
// respond with "Hello World!" on the homepage
app.get('/', function (req, res) {
  res.send('Hello World!');
})

// accept POST request on the homepage
app.post('/', function (req, res) {
  res.send('Got a POST request');
})
```

PATH

CALLBACK or HANDLER

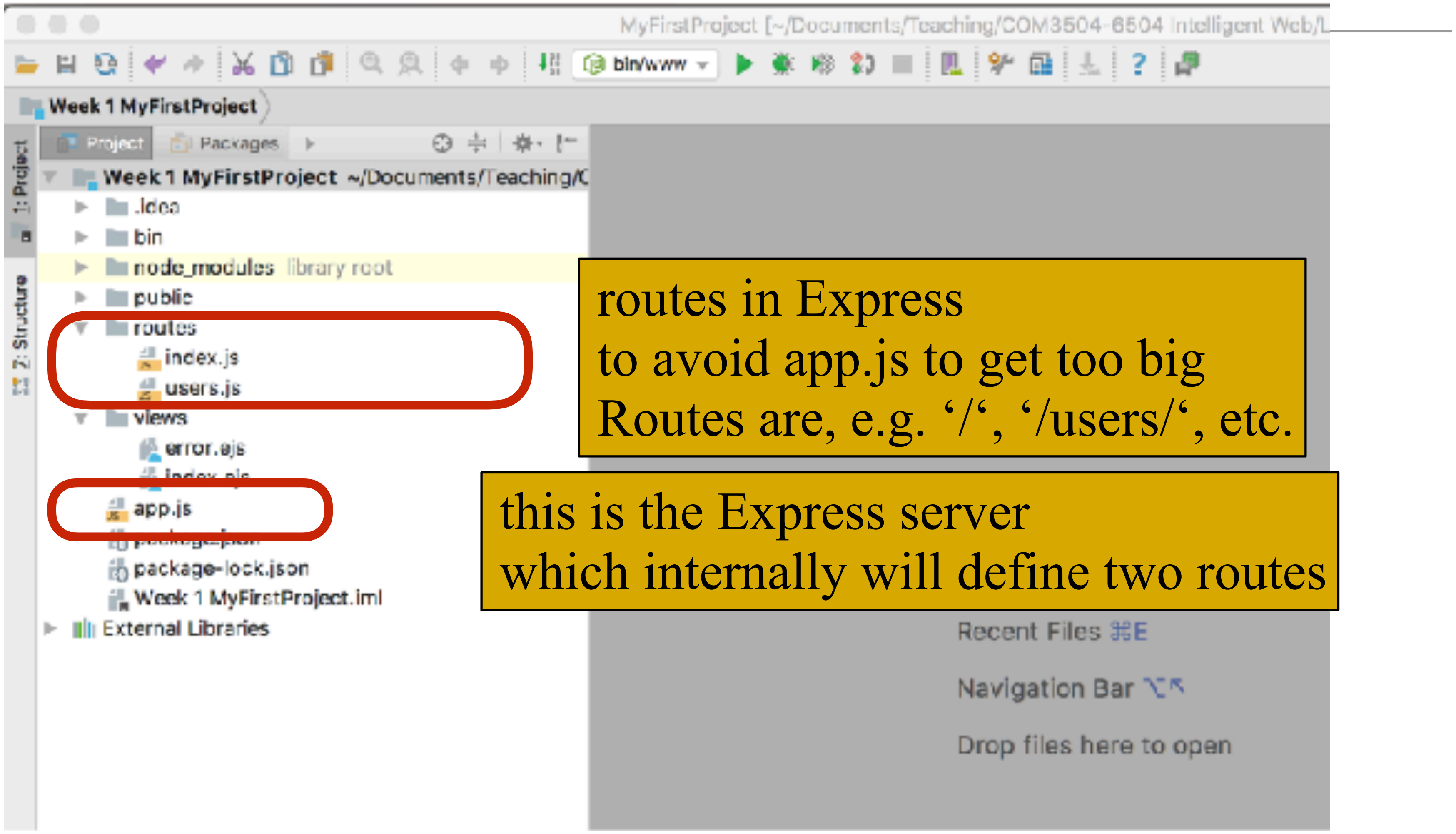
- app is an instance of express
- is an HTTP request method (POST, GET)
- PATH is a path on the server,
- HANDLER is the callback function execute

app.all

- Special routing method not derived from any HTTP method
- Used for representing all request methods.

```
// respond with "Hello World!" to all type of
// requests (post, get, etc.) on the homepage

app.all('/', function (req, res, next) {
  res.send('Got a request');
})
```

The server side in Express



routes in Express
to avoid app.js to get too big
Routes are, e.g. '/', '/users/', etc.

this is the Express server
which internally will define two routes

The University Of Sheffield.

Route Paths

- Route paths define the endpoints at which requests can be made to.
  - e.g. '/', or '/users/' …
- In express they can be:
  - strings
  - string patterns
  - regular expressions.
- Note!
  - Query strings are not a part of the route path.
    - In http://localhost/index.html**?index=34**
    - **?index=34** is **not** part of the route path

Examples

```
// with match request to the root
app.get('/', function (req, res) {
  res.send(‘root requested')
})

// will match requests to /about
app.get('/about', function (req, res) {
  res.send(‘about requested')
})

// will match request to /random.html
app.get('/random.html', function (req, res) {
  res.send(‘random.html requested')
})
```

String Patterns

```
// will match acd and abcd
app.get('/ab?cd', function(req, res) {
  res.send('ab?cd')
})

// will match abcd, abbcd, abbbcd, and so on
app.get('/ab+cd', function(req, res) {
  res.send('ab+cd')
})

// will match abcd, abxcd, abRABDOMcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
  res.send('ab*cd')
})

// will match /abe and /abcde
app.get('/ab(cd)?e', function(req, res) {
 res.send('ab(cd)?e')
})
```

Note! The characters **?, +, and ()** are subsets of their Regular Expression counterparts. The hyphen (-) and the dot (.) are interpreted literally by string-based paths. * means any char

The University Of Sheffield.

Regular Expressions

- Following the Unix standard (also used in the vim editor)

```
// will match anything with an a in the route name:
app.get(/a/, function(req, res) {
  res.send('/a/')
})

// will match butterfly, dragonfly; but not butterflyman,
dragonfly man, and so on
app.get(/.*fly$/, function(req, res) {
  res.send('/.*fly$/')
})
```

Defining routes

- Route handlers for a single route path can be created using

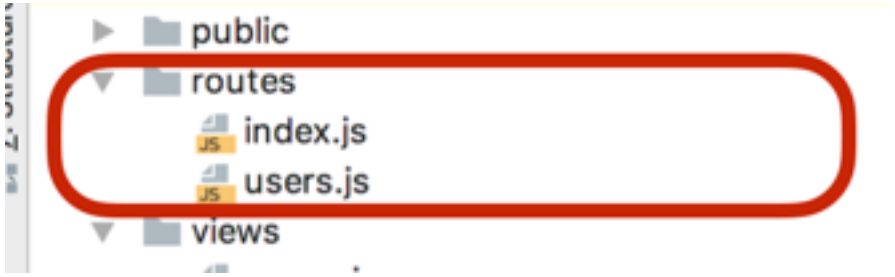
```
app.route('/book')

  .get(function(req, res) {
    res.send('Get a random book');
  })

  .post(function(req, res) {
    res.send('Add a book');
  })
```

- This reduce redundancy and typos.

In IntelliJ

- in app.js

```
var users = require('./routes/users');
…
app.use('/users', users);
```



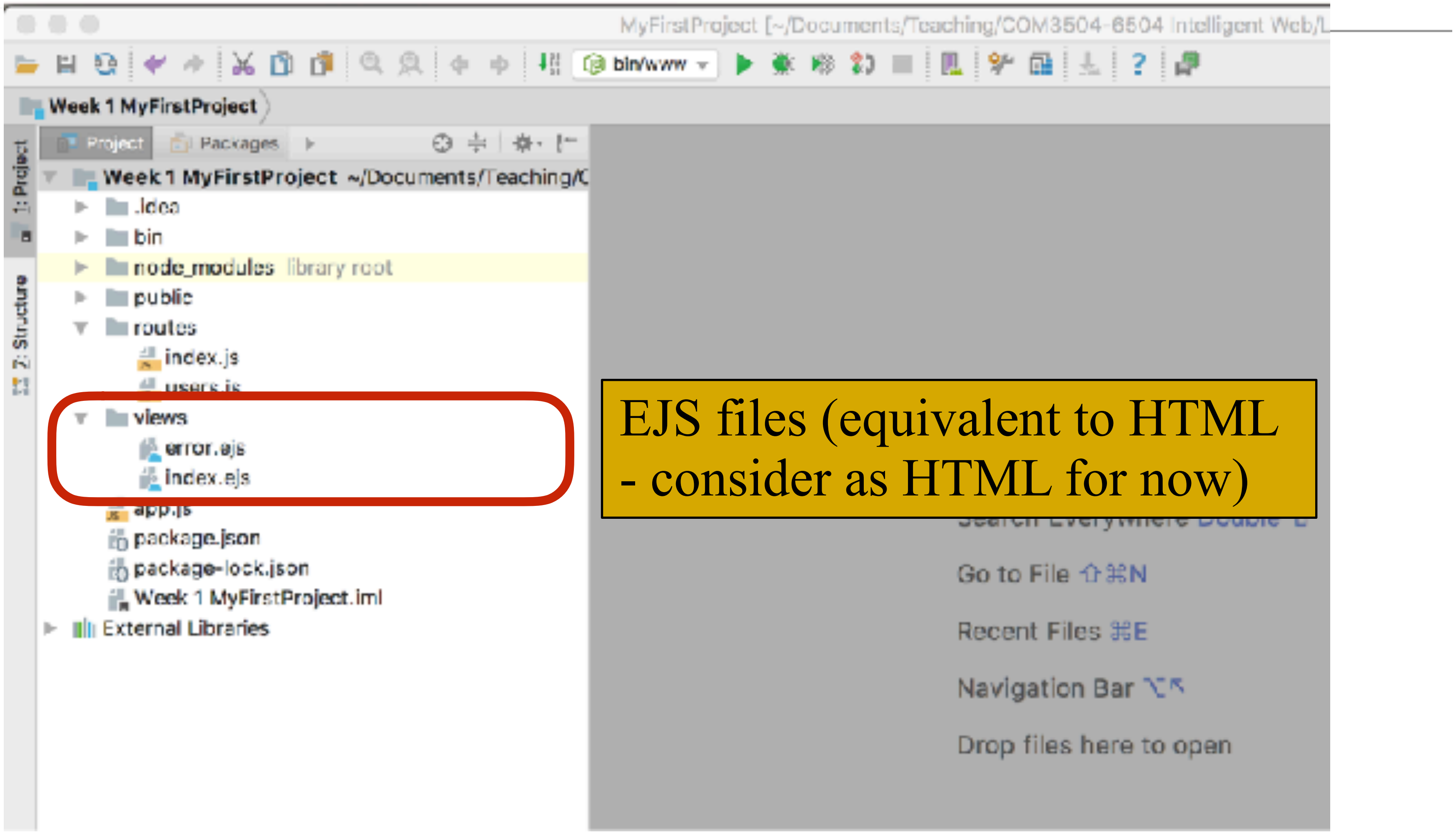
- in e.g. routes/users.js

```
/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('whatever');
});
```

- `this will respond to http://localhost:3000/users/`

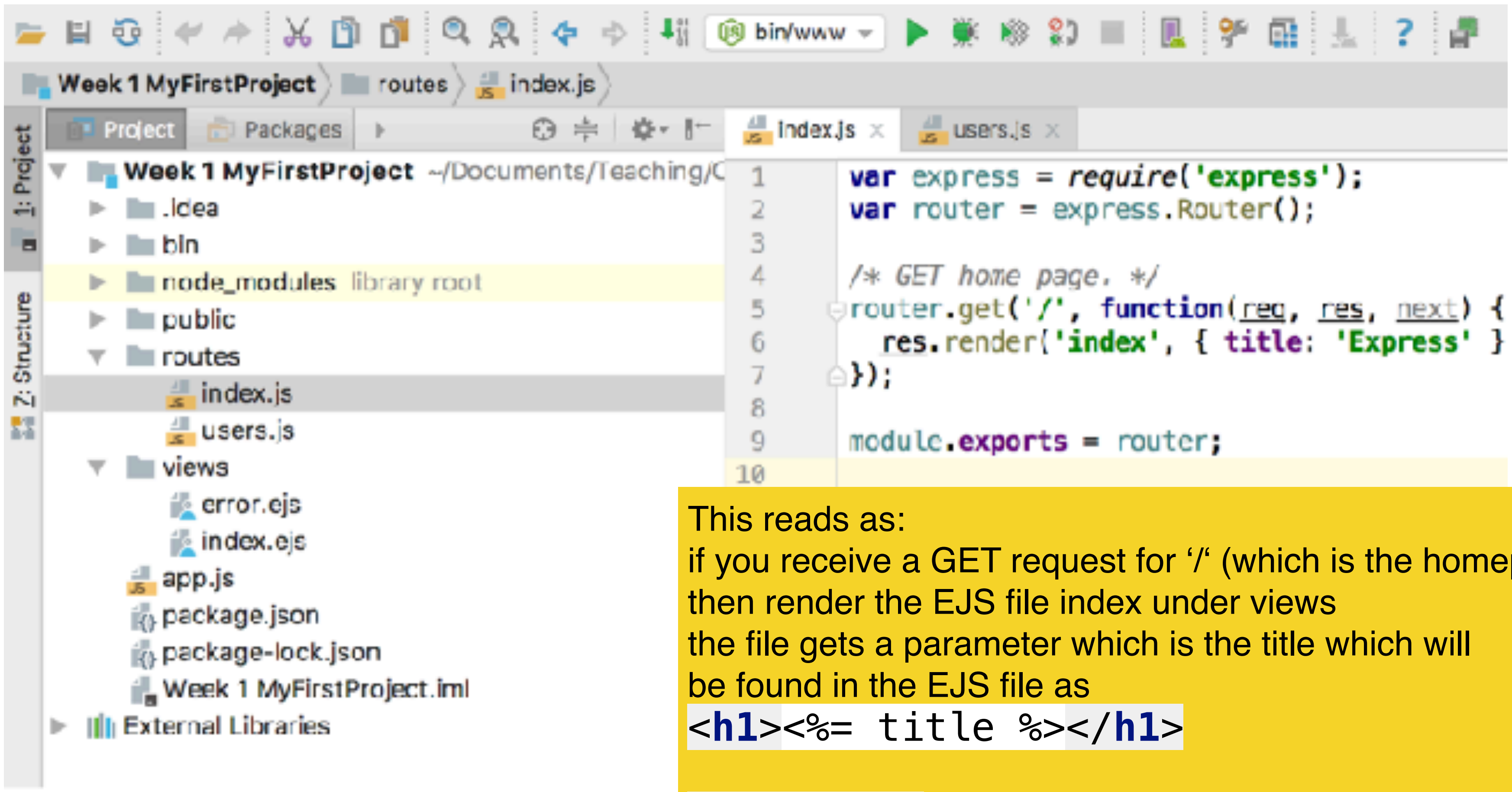
The client side

EJS Template files



EJS files (equivalent to HTML - consider as HTML for now)

Serving EJS Template files in routes

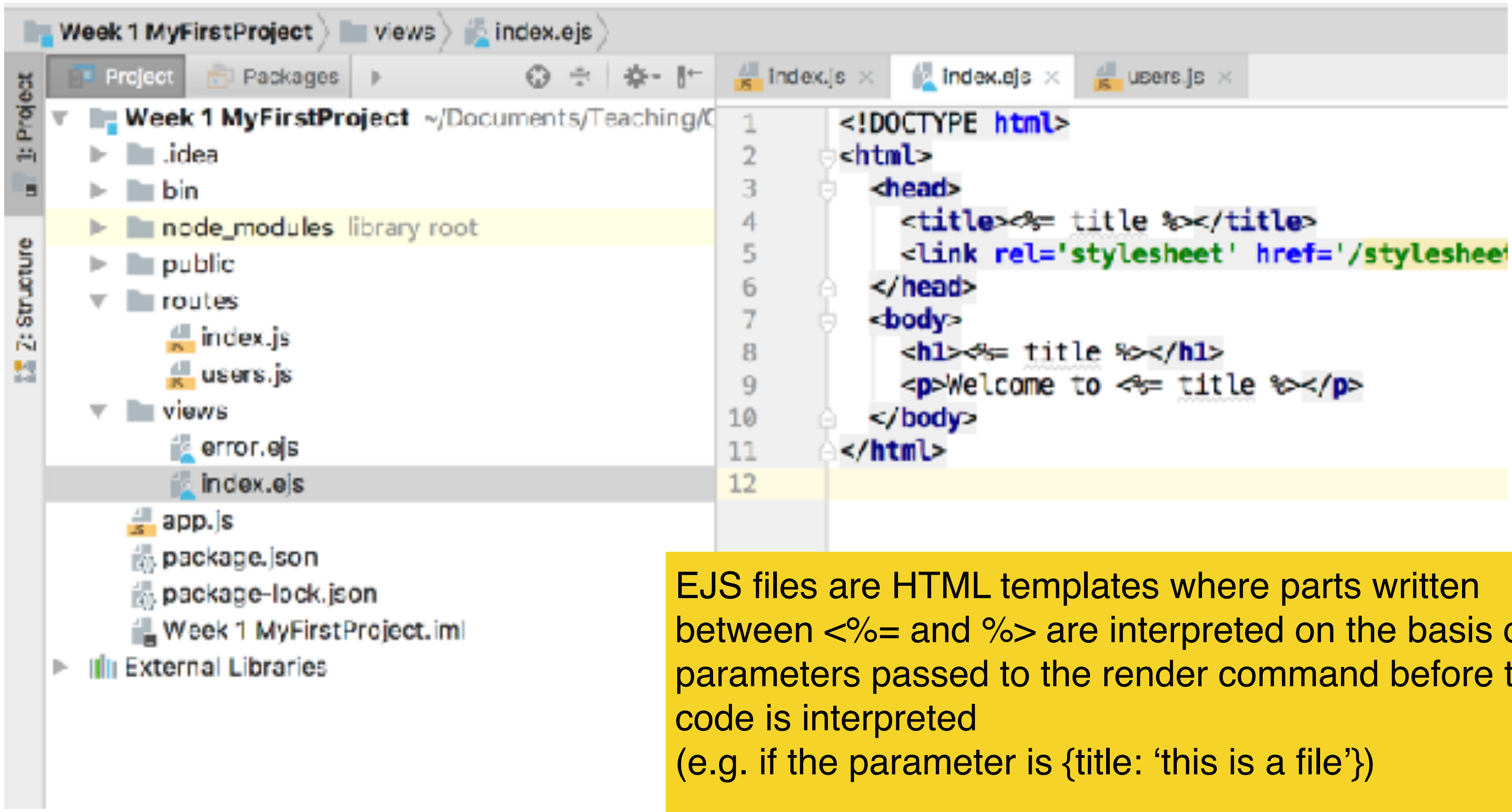


```
var express = require('express');
var router = express.Router();

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

module.exports = router;
```

This reads as:
if you receive a GET request for '/' (which is the homepage)
then render the EJS file index under views
the file gets a parameter which is the title which will
be found in the EJS file as

```
<h1><%= title %></h1>
```

{title: 'Express'}
is a parameter passed to the file so that it is replaced into the code BEFORE the html code is interpreted


Week 1 MyFirstProject › views › index.ejs

Project Packages

- Week 1 MyFirstProject ~/Documents/Teaching/C
  - .idea
  - bin
  - node_modules library root
  - public
  - routes
    - index.js
    - users.js
  - views
    - error.ejs
    - index.ejs
  - app.js
  - package.json
  - package-lock.json
  - Week 1 MyFirstProject.iml
- External Libraries

index.js | index.ejs | users.js

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/styleshee
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

EJS files are HTML templates where parts written between <%= and %> are interpreted on the basis of the parameters passed to the render command before the HTML code is interpreted
(e.g. if the parameter is {title: ‘this is a file’})

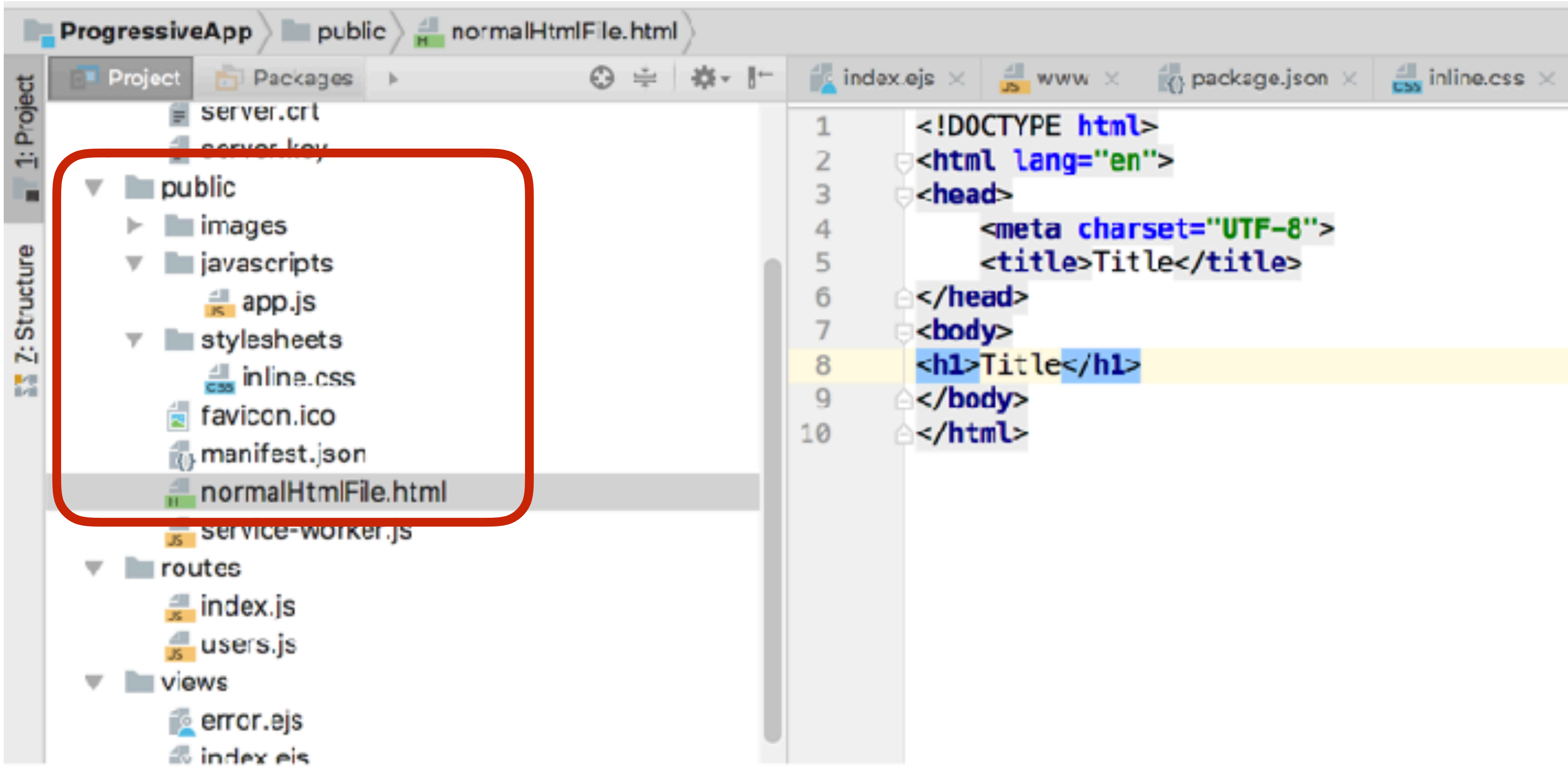
`<h1><%= title %></h1>`

will be passed to the HTML interpreter as

`<h1>This is a file</h1>`

How to serve static files

- If no special rendering is needed, you can insert HTML files under the public directory



ProgressiveApp > public > normalHtmlFile.html

```
server.crt
server.key
public
  images
  javascripts
    app.js
  stylesheets
    inline.css
  favicon.ico
  manifest.json
  normalHtmlFile.html
  service-worker.js
routes
  index.js
  users.js
views
  error.ejs
  index.ejs
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>Title</h1>
</body>
</html>
```

Serving static files

http://expressjs.com/starter/static-files.html

- Serving static files is accomplished with the help of a built-in middleware in Express
  - express.static.
- Pass the name of the directory where you keep you static files
  - For example, if you keep your images, CSS, and JavaScript files in a directory named **public**, you can do this:

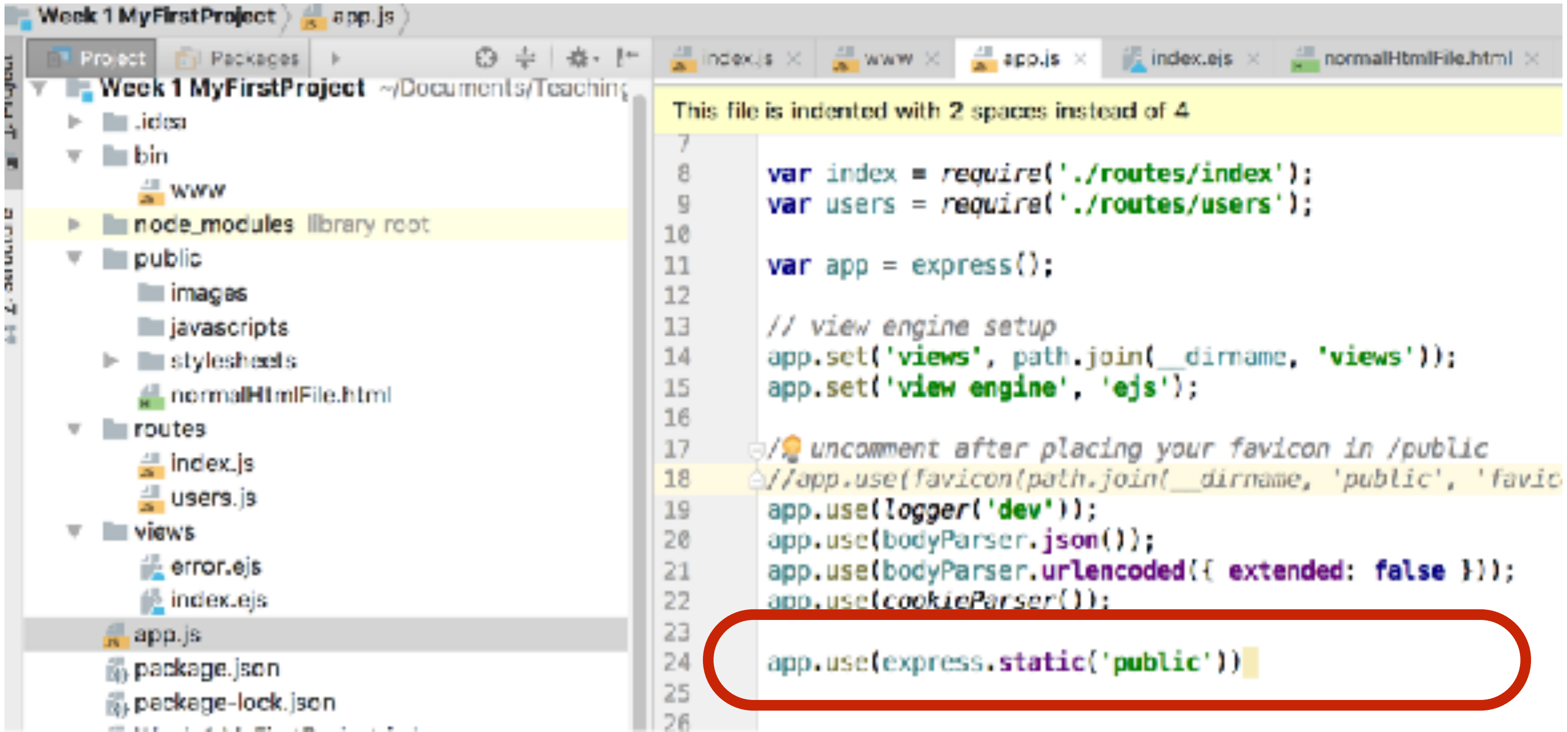
```
app.use(express.static('public'));
```

note!

public files

- static files requested by GET are returned automatically by express (no need for specific paths in index.js)
  - then add this line in app.js

```
app.use(express.static('public'))
```



```
var index = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favic
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());

app.use(express.static('public'))
```

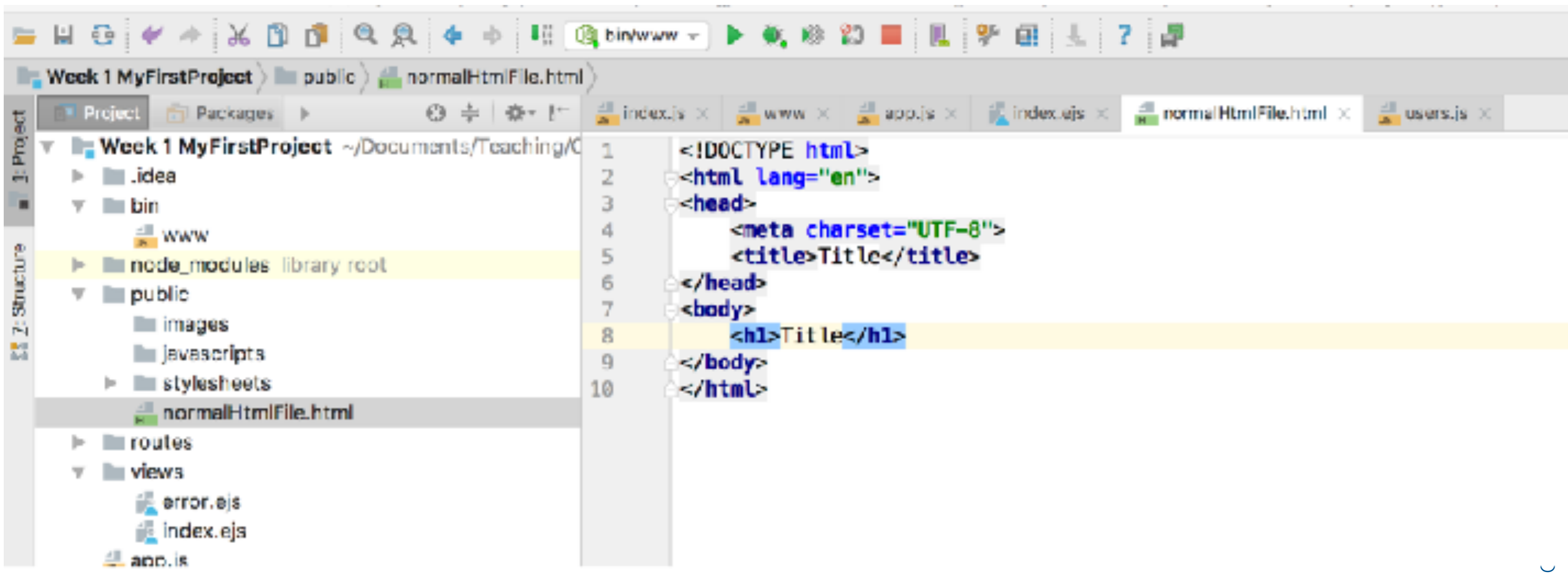
Static

- Now, you will be able to load ALL files under the public directory:

- http://localhost:3000/images/kitten.jpg
- http://localhost:3000/css/style.css
- http://localhost:3000/js/app.js
- http://localhost:3000/images/bg.png
- http://localhost:3000/hello.html

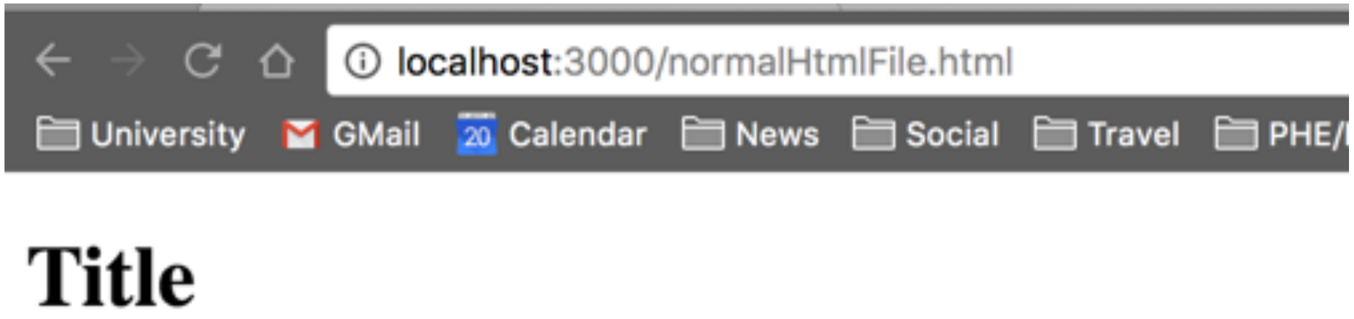
create a file under public

- Right click on the public folder and choose 'new'. Choose new HTML file



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>Title</h1>
</body>
</html>
```

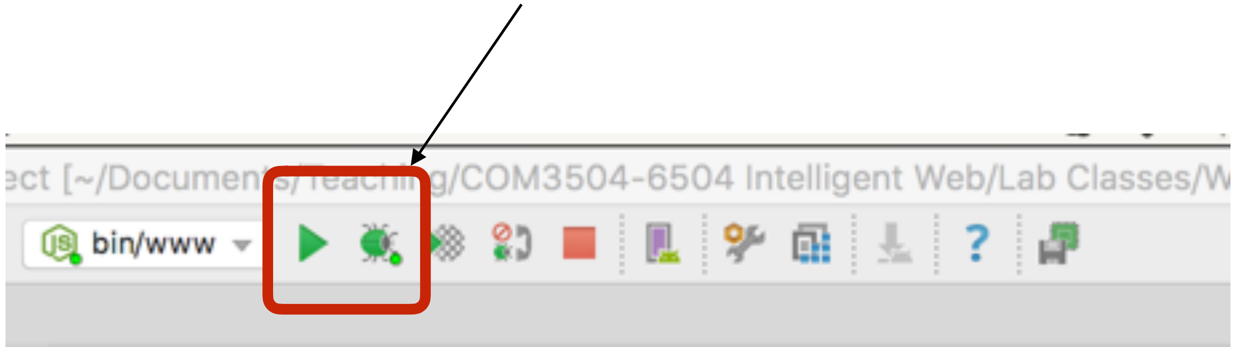
on Chrome: go to http://localhost:3000/normalHtmlFile.html



Title

Changes?

- Note: changes to the code have different effects:
  - changes to the client
    - Views and Public directories
    - just require reloading the page in the browser
  - changes to the server (node js) requires restarting the server from IntelliJ


ect [~/Documents/Teaching/COM3504-6504 Intelligent Web/Lab Classes/W
bin/www

Contacting other servers from node.js

Posting requests from node.js

- We may need to post a request to another server in order to retrieve further information
  - Post a query to a database
  - Query the Twitter API
  - …
- Two features of Node.js are useful:
  - It is Javascript
  - It is event-driven

GETting from node.js server

http://samwize.com/2013/08/31/simple-http-get-slash-post-request-in-node-dot-js/

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

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

// Set the headers
var headers = {
    'User-Agent':       'Super Agent/0.0.1',
    'Content-Type':     'application/x-www-form-urlencoded'
}

// Configure the request
var options = {
    url: 'http://samwize.com',
    method: 'GET',
    headers: headers,
    qs: {'key1': 'xxx', 'key2': 'yyy'}
}

// Start the request
request(options,
  function (error, response, body) {
    if (!error && response.statusCode == 200) {
        // Print out the response body
        console.log(body)
    }
})
```

HTTP header

Parameters for the GET

Results are here

Callback function (called when results are received)

POSTing from node.js server

http://samwize.com/2013/08/31/simple-http-get-slash-post-request-in-node-dot-js/

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

```
var request = require('request');
…
app.post('/request_from_form', function

    // Set the headers
    var headers = {
        'User-Agent':       'Super Agent/0.0.1',
        'Content-Type':     'application/x-www-form-urlencoded'
    }
    // Configure the request
   var options = {
     url: 'http://samwize.com',
     method: 'POST',
     headers: headers,
     form: {'key1': 'xxx', 'key2': 'yyy'}
   }
   // Start the request
   request(options,
      function (error, response, body) {
        if (!error && response.statusCode == 200) {
            // Print out the response body
            console.log(body)
     }
```

HTTP header

Parameters for the POST

Callback function (called when results are received)

Connecting node.js to MySQL

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.

Creating a node.js module

Modules

https://quickleft.com/blog/creating-and-publishing-a-node-js-module/

- As mentioned modules are equivalent to Java packages
- They are used to create sets of functionalities around an “object”
- To create a module, type
    - npm init
- This will ask you a number of questions and generate a package.json file as output
    - more on json next week

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (nodeTest) intelligent_web_module
version: (1.0.0)
description: this is the first node.js module we create
entry point: (index.js)
test command:
git repository:
keywords:
author: Fabio Ciravegna
license: (ISC) MIT
About to write to /Users/fabio/Documents/Programs/Android/nodeTest/package.json:
{
  "name": "intelligent_web_module",
  "version": "1.0.0",
  "description": "this is the first node.js module we create",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Fabio Ciravegna",
  "license": "MIT"
}

Is this ok? (yes)
```

- The primary 2 items that we are concerned with here are
  - require
    - You require other modules that you wish to use in your code (this is equivalent to the include in Java)
  - exports
    - your module exports anything that should be exposed publicly.
- For example:

```
var other = require('other_module');
module.exports = function() {
    console.log(other.doSomething());
}
```

index.js

```
/**
 * Escape special characters in the given string of html.
 *
 * @param  {String} html
 * @return {String}
 */
module.exports = {
  escape: function(html) {
    return String(html).replace(/&/g, '&amp;').replace(/"/g, '&quot;')
      .replace(/'/g, '&#39;').replace(/</g, '&lt;').replace(/>/g, '&gt;');
  },

  /**
   * Unescape special characters in the given string of html.
   *
   * @param  {String} html
   * @return {String}
   */
  unescape: function(html) {
    return String(html).replace(/&amp;/g, '&').replace(/&quot;/g, '"')
      .replace(/&#39;/g, ''').replace(/&lt;/g, '<').replace(/&gt;/g, '>');
  }
};
```

Usage

- Create a server folder
- create a subfolder called ‘node_modules’
  - required
  - move your module there (suppose it is called ‘intelligent_web_module’)
- Create a file called server.js in the server folder containing

```
var iwm= require (‘intelligent_web_module')
var myString= ‘this is not a string < at least I do not think so >’;
var escapedString= ivm.escape(myString);

console.log (escapedString);
```

- run node server.js

```
it will print ‘this is not a string &lt at least I do not think so &gt’
```

Debugging node.js

Debugging node.js

- Node.js does not run in a browser
  - it runs on a server

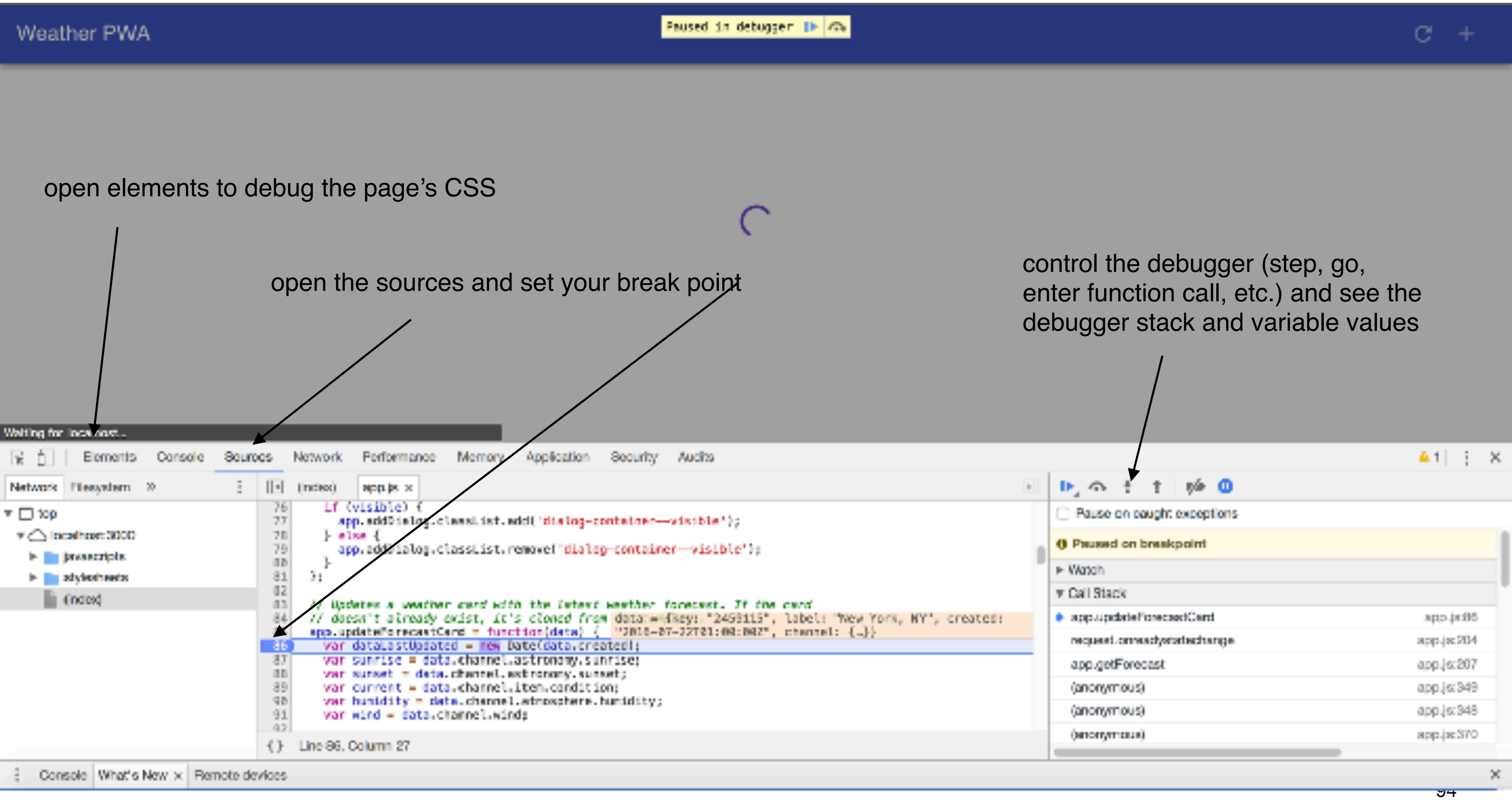
Debugging HTML/Javascript in Chrome

https://developer.chrome.com/devtools/docs/javascript-debugging

- We will be using IntelliJ
  - if you have used AndroidStudio or WebStorm, it is the same product with different flavours
- There are two parts in any client/server architecture:
  - the client (e.g. a browser): this can be debugged with Chrome
  - the server (the nodejs server), this is to be debugged with Intellij

Chrome debugging (client)

- open from right menu (or view>developers>javascript console on a Mac)



open elements to debug the page's CSS

open the sources and set your break point

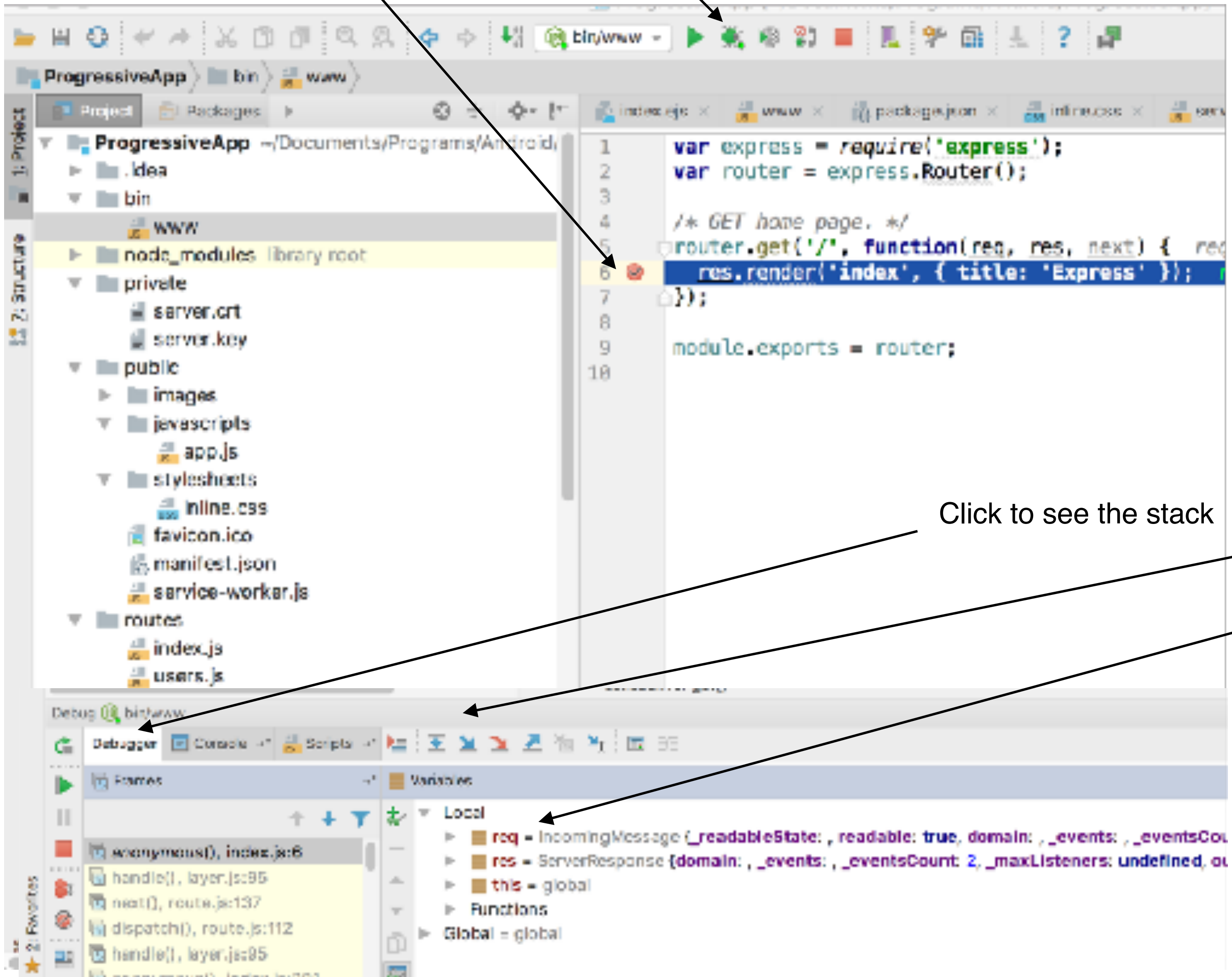
control the debugger (step, go, enter function call, etc.) and see the debugger stack and variable values

```
    if (visible) {
      app.addDialog.classList.add('dialog-container--visible');
    } else {
      app.addDialog.classList.remove('dialog-container--visible');
    }
  };

  // Updates a weather card with the latest weather forecast. If the card
  // doesn't already exist, it's cloned from  data = {key: "2459115", label: "New York, NY", created:
  app.updateForecastCard = function(data) {  "2016-07-22T21:00:00Z", channel: {…}}
    var dataLastUpdated = new Date(data.created);
    var sunrise = data.channel.astronomy.sunrise;
    var sunset = data.channel.astronomy.sunset;
    var current = data.channel.item.condition;
    var humidity = data.channel.atmosphere.humidity;
    var wind = data.channel.wind;
```

Debugging the server

Click to set a break point

Click to debug the server



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

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

module.exports = router;
```

```
anonymous(), index.js:6
handle(), layer.js:95
next(), route.js:137
dispatch(), route.js:112
handle(), layer.js:95
```

```
Local
  req = IncomingMessage {_readableState: , readable: true, domain: , _events: , _eventsCou
  res = ServerResponse {domain: , _events: , _eventsCount: 2, _maxListeners: undefined, ou
  this = global
  Functions
Global = global
```

Click to see the stack

control the debugger (step, go, enter function call, etc.)

local variables and stack

Questions?

We will continue in teh lab this afternoon