Galileo Tutorial
Networking and node.js

Senzations 2014

Jason Wright

Biograd na Moru

intel Galileo

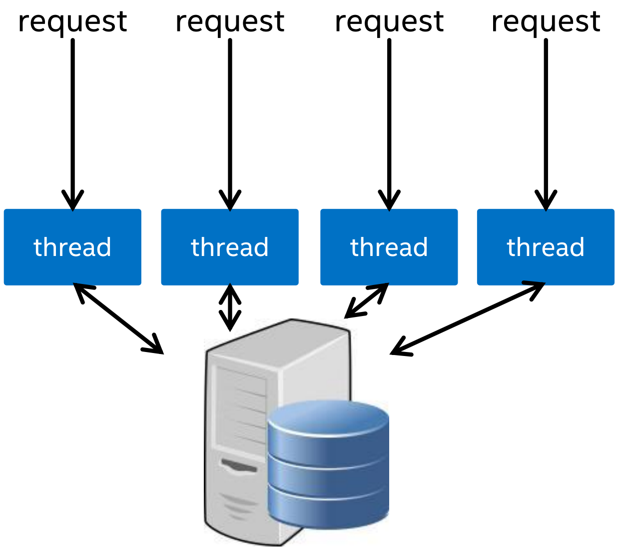
What will you make?

Learning goals

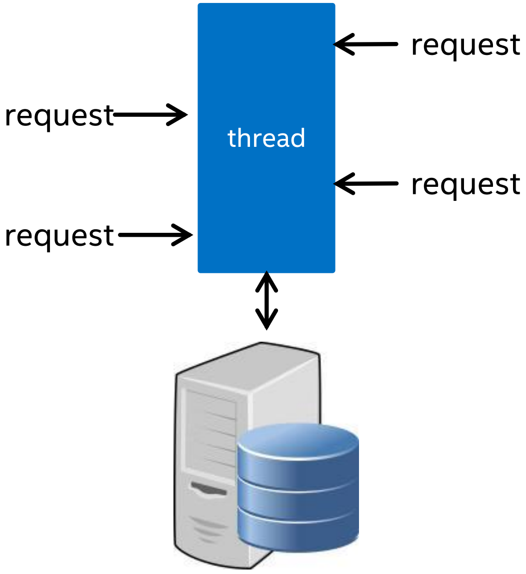
- Basics of node.js
  - why & how it’s useful
  - server/client networking
- How to deploy a node.js server on Galileo
- Interacting with Galileo through the browser
- Reading and displaying sensor data

What is node.js?

- node.js (or just *node*) is a JavaScript runtime designed for lightweight server applications
- It is not a full webserver (e.g. Apache, nginx)
- The incoming request handler is single-threaded instead of multi-threaded



Traditional server



node.js server

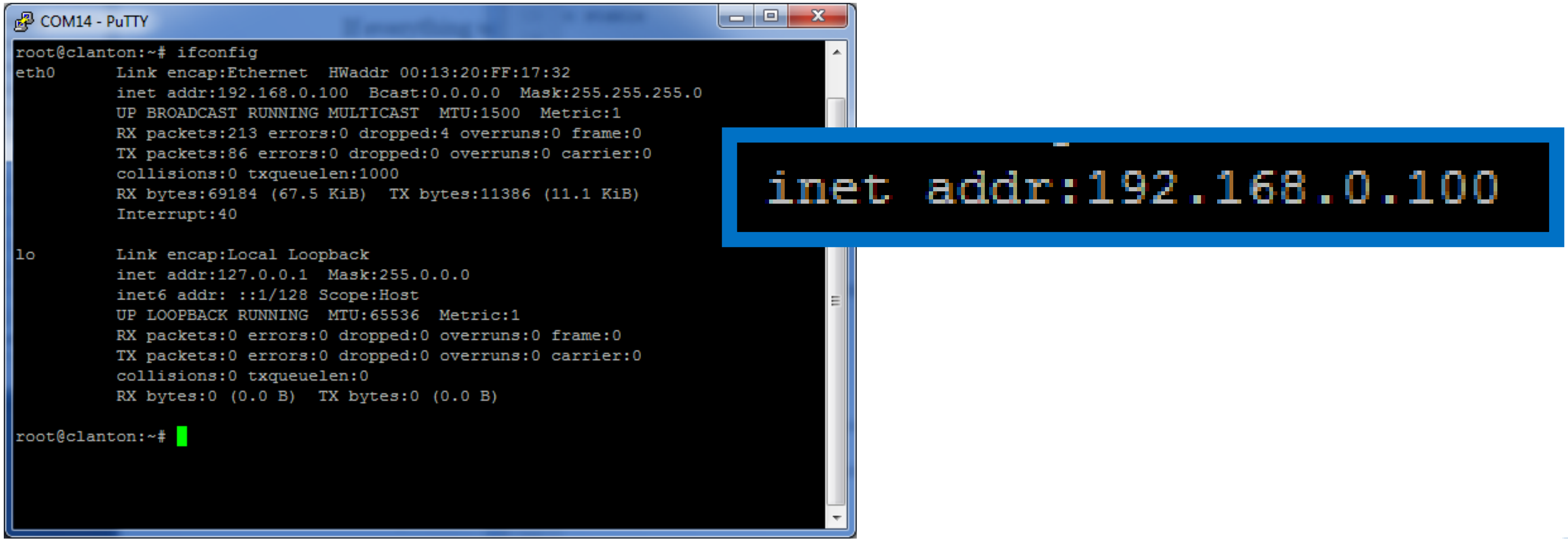
Why use node.js?

- **Consistent:** Server/client language and data representations are the same
- **Scalable:** Single-threaded architecture minimizes memory usage and avoids cost of context-switching between threads
  - Problem: What if one client is computationally demanding?
  - Problem: What if there's a core exception?
- **Fast** (at certain things)
  - node is especially useful if I/O is likely to be your bottleneck (i.e., your server isn't doing that much)
  - examples: queued inputs, data streaming, web sockets

Generally speaking, node is ideal for *lightweight*, *real-time* tasks and a bad choice for *computationally intensive* tasks

Setting up networking on Galileo

- Setup WiFi SSID and authentication (as needed)
- Enable network interface
  - **`ifup eth0`**
  - **`ifup wlan0`**
- Check network status
  - **`ifconfig`**

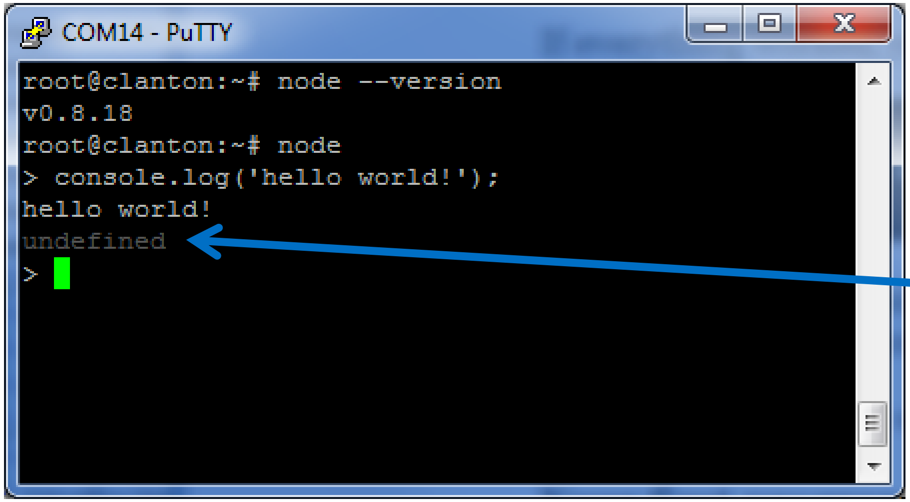


```
root@clanton:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:13:20:FF:17:32
          inet addr:192.168.0.100  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:213 errors:0 dropped:4 overruns:0 frame:0
          TX packets:86 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:69184 (67.5 KiB)  TX bytes:11386 (11.1 KiB)
          Interrupt:40

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@clanton:~#
```

```
inet addr:192.168.0.100
```

Setting up node.js on Galileo

- Node should come preinstalled on your Galileo image
- Verify that it's installed and working:
  - **`node --version`**
- Test it out using the interactive shell:



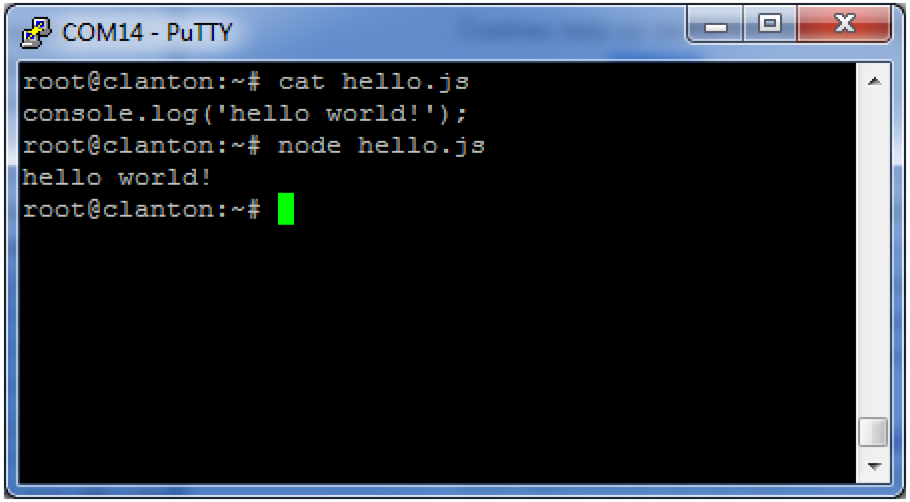
```
root@clanton:~# node --version
v0.8.18
root@clanton:~# node
> console.log('hello world!');
hello world!
undefined
> 
```

The interactive shell will always print the return value of any command. `console.log` has no return value, so it prints '`undefined`'.

- Ctrl+D to quit (or Ctrl+C twice)

Setting up node.js on Galileo

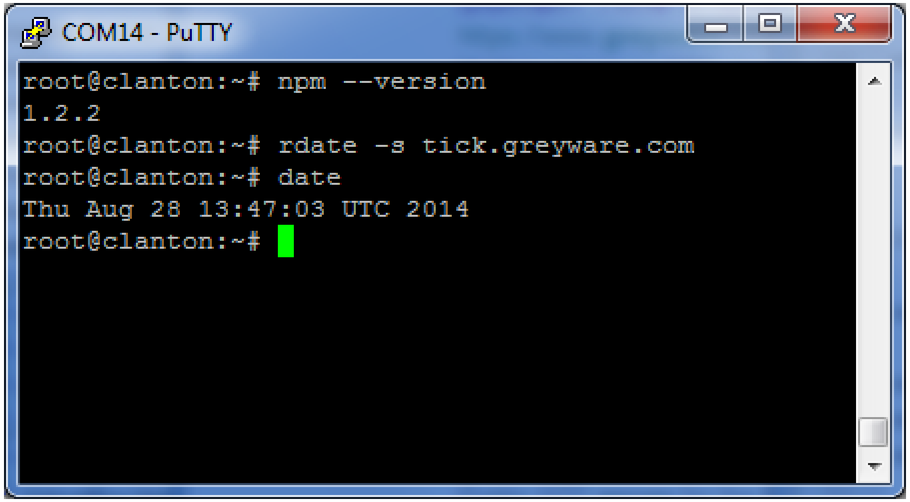
- You can also run node by passing in a script – this is how we'll handle server operations



```
root@clanton:~# cat hello.js
console.log('hello world!');
root@clanton:~# node hello.js
hello world!
root@clanton:~#
```

Setting up node.js on Galileo

- Node comes with its own package manager, called **npm**
- Also verify that this is installed and working
  - **`npm –version && npm ls`**
- We'll install some packages that will come in handy, but first we need to correct the Galileo's clock

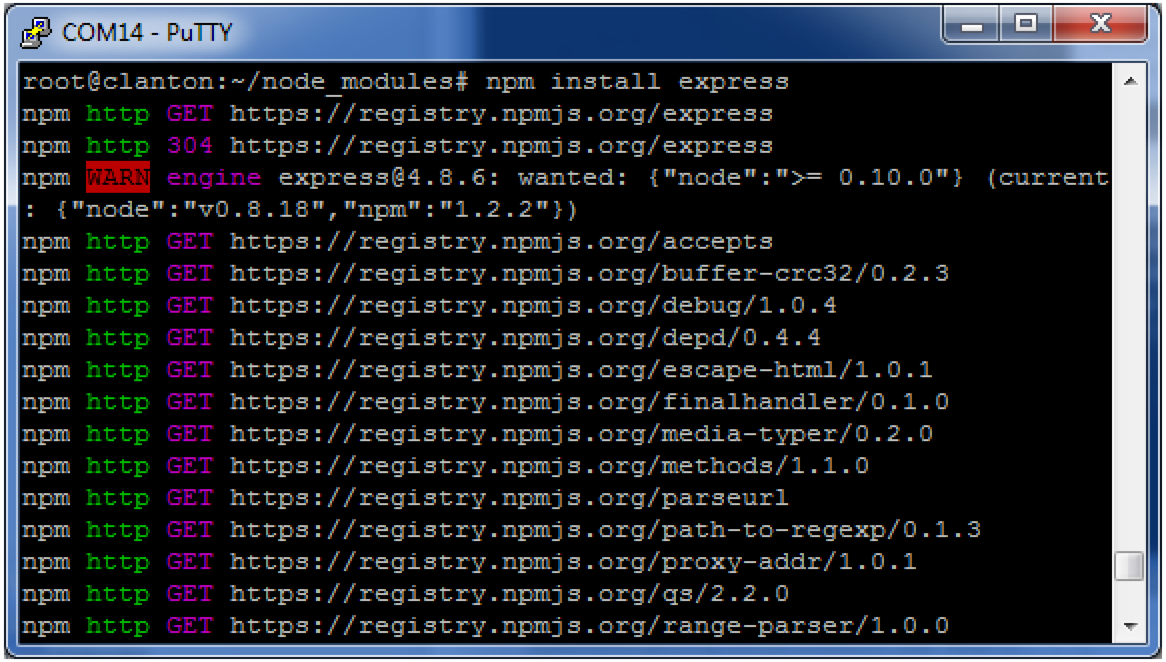


```
root@clanton:~# npm --version
1.2.2
root@clanton:~# rdate -s tick.greyware.com
root@clanton:~# date
Thu Aug 28 13:47:03 UTC 2014
root@clanton:~#
```

  - If you don't do this, "date" will be wrong and npm installations will fail with an SSL CERT_NOT_YET_VALID error

Setting up node.js on Galileo

- Install a few modules (this can be a bit slow...)
  - **`npm install express ejs socket.io galileo-io`**



```
root@clanton:~/node_modules# npm install express
npm http GET https://registry.npmjs.org/express
npm http 304 https://registry.npmjs.org/express
npm WARN engine express@4.8.6: wanted: {"node":">= 0.10.0"} (current
: {"node":"v0.8.18","npm":"1.2.2"})
npm http GET https://registry.npmjs.org/accepts
npm http GET https://registry.npmjs.org/buffer-crc32/0.2.3
npm http GET https://registry.npmjs.org/debug/1.0.4
npm http GET https://registry.npmjs.org/depd/0.4.4
npm http GET https://registry.npmjs.org/escape-html/1.0.1
npm http GET https://registry.npmjs.org/finalhandler/0.1.0
npm http GET https://registry.npmjs.org/media-typer/0.2.0
npm http GET https://registry.npmjs.org/methods/1.1.0
npm http GET https://registry.npmjs.org/parseurl
npm http GET https://registry.npmjs.org/path-to-regexp/0.1.3
npm http GET https://registry.npmjs.org/proxy-addr/1.0.1
npm http GET https://registry.npmjs.org/qs/2.2.0
npm http GET https://registry.npmjs.org/range-parser/1.0.0
```

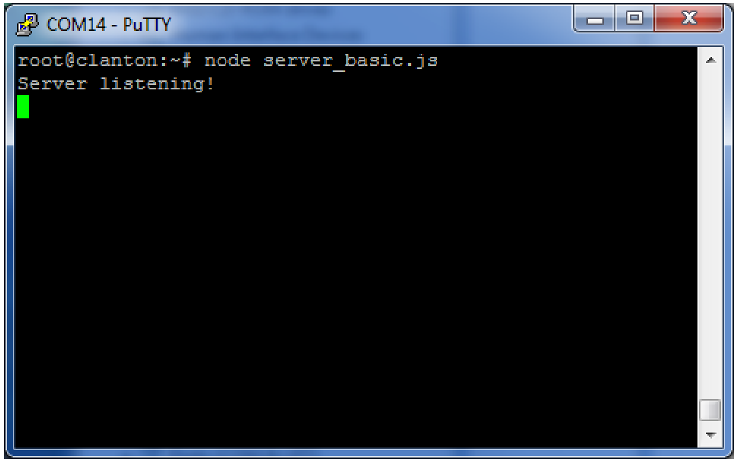
Writing a server, part 1: http

- The **http** module handles *requests* and *responses* via a *server* object
- Most basic example (server_basic.js):

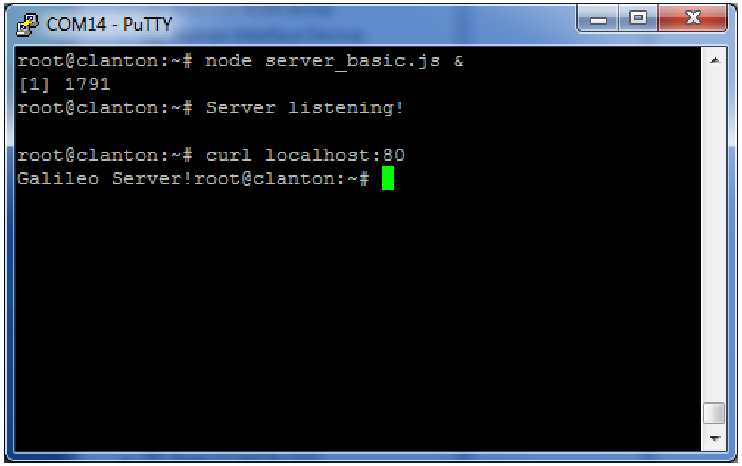
```
var http = require("http");
var server = http.createServer(function(request, response) {
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("Galileo Server!");
    response.end();
});

server.listen(80);
console.log("Server listening!");
```

- Launch this server by calling it with node:
  - `node server_basic.js`
- You'll notice the program doesn't terminate—it will continually run and process requests as they come in

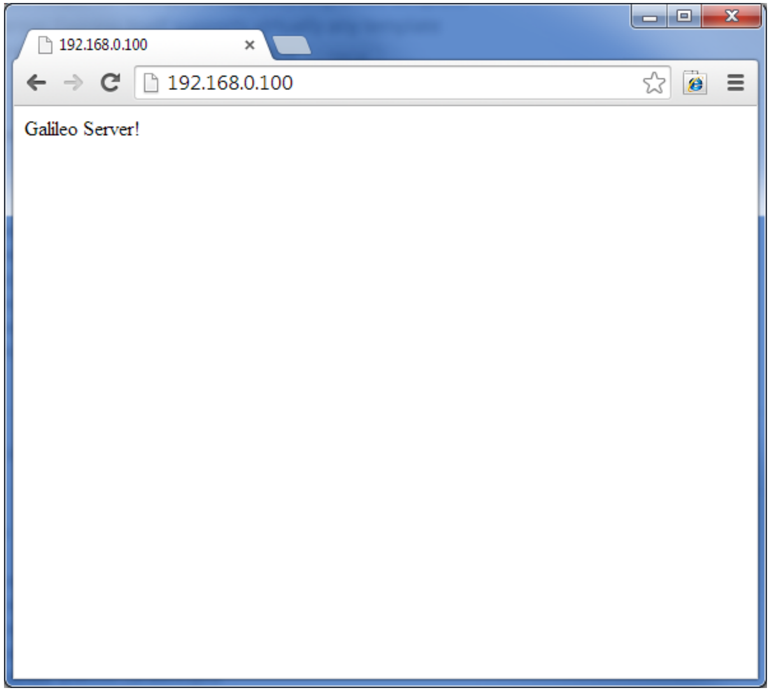
Writing a server, part 1: http



```
root@clanton:~# node server_basic.js
Server listening!
```



```
root@clanton:~# node server_basic.js &
[1] 1791
root@clanton:~# Server listening!

root@clanton:~# curl localhost:80
Galileo Server!root@clanton:~#
```



Galileo Server!

Writing a server, part 2: express

- **express** is a web application framework for node
  - In this context we'll mainly be using it as a way to serve up dynamically generated HTML content, but it has many other features
- Main benefit in this context is that we can use a templating engine to avoid spitting out tons of HTML in a redundant way
  - We'll use *ejs* as our templating engine (*jade*, *haml* are also popular)
  - Instead of directly writing the HTTP response, pass in a view and a set of parameters

Writing a server, part 2: express

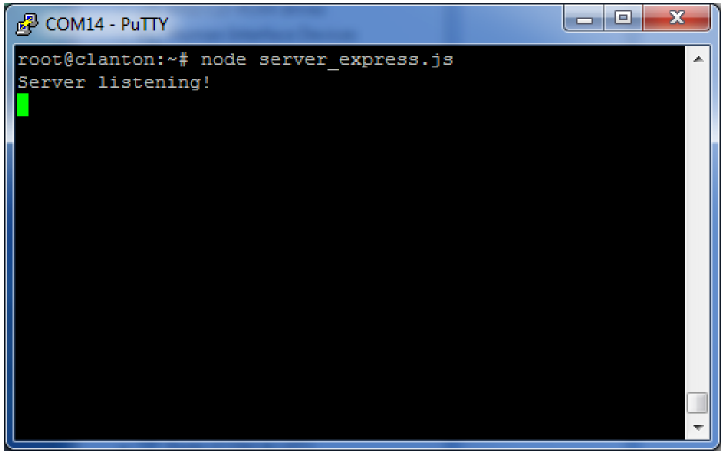
- Server (server_express.js):

```
var express = require('express');
var app = express();
app.set('view engine', 'ejs');
app.get('/', function(request, response) {
    response.render('index', {
        title: 'Home',
        message: 'This is an Express app running on the Galileo'
    });
});
app.listen(80);
console.log("Server listening!");
```

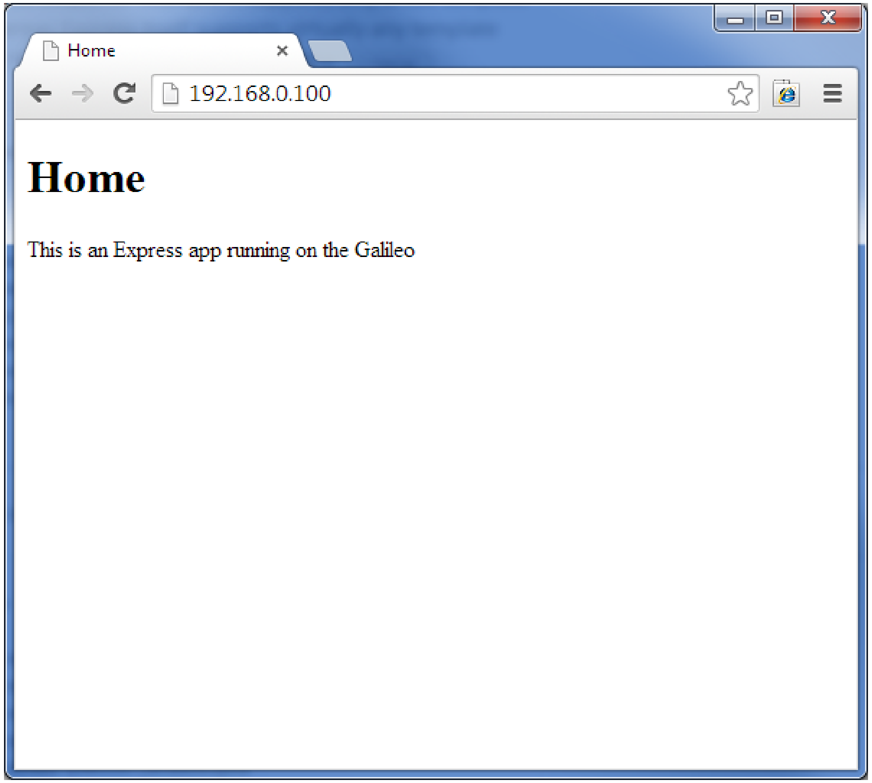
- Template (views/index.ejs):

```
<!doctype html>
<html lang="en">
<head>
  <title><%= title %></title>
</head>
<body>
<h1><%= title %></h1>
<p><%= message %></p>
</body>
</html>
```

Writing a server, part 2: express



```
root@clanton:~# node server_express.js
Server listening!
```



Home

This is an Express app running on the Galileo

Writing a server, part 3: sockets

- WebSockets – a full-duplex (bidirectional) TCP communications channel
- socket.io – a simple-to-use WebSockets implementation for node
- server_socket.js:

```
var io = require('socket.io').listen(server);
...
io.on('connection', function(socket) {
    console.log('user connected');
    socket.on('myAction', function(msg) {
        console.log('woohoo!');
    });
});
```

- views/action.ejs:

```
<script src="/socket.io/socket.io.js"></script>
<script>var socket = io();</script>
…
<button onclick="socket.emit('myAction');">Click Me!</button>
```

Working with sensor data

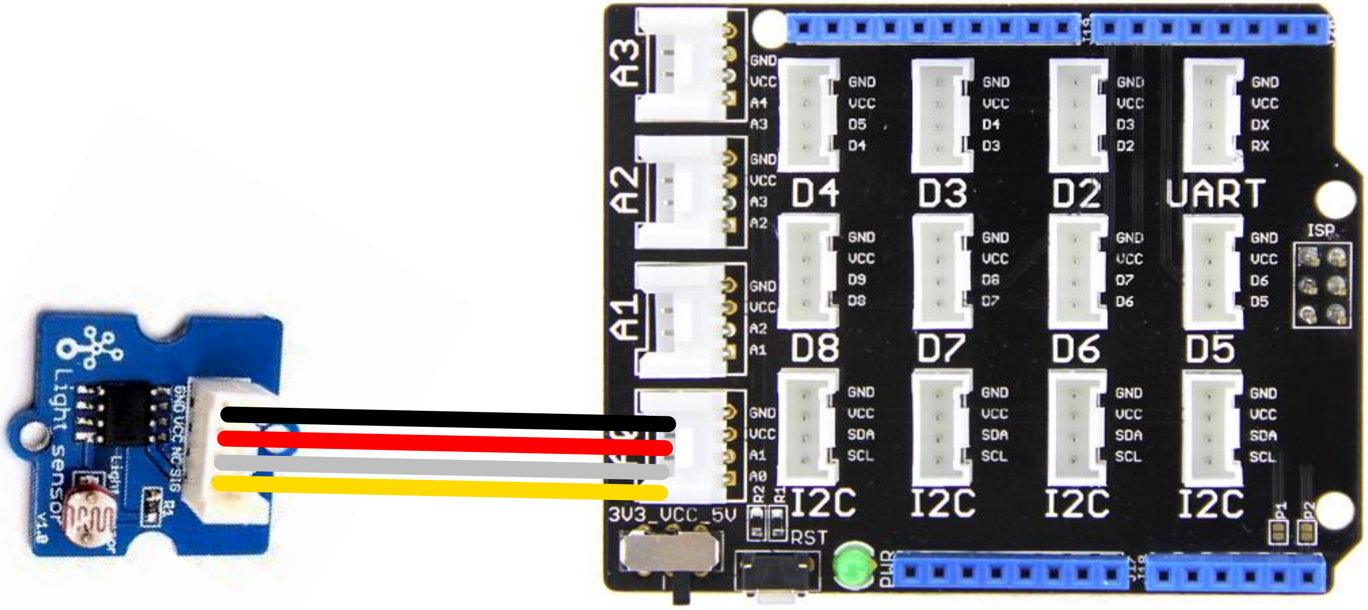
- Linux provides a virtual filesystem called **sysfs** that allows for easy access to underlying hardware from userspace
- This makes working with sensor data as simple as reading and writing to files
  - Arduino functionality on Galileo is implemented via abstracted sysfs interactions
- Quick example: reading the core temperature
  - `cat /sys/class/thermal/thermal_zone0/temp`
  - Divide by 1000 to get the SoC temperature in °C
  - (Quark can run hot, but it's normal)

Working with sensor data – GPIO access

- Export the port
  - `echo -n "3" > /sys/class/gpio/export`
  - A new folder (gpio3) will appear in /sys/class/gpio
  - This particular GPIO pin is wired to the green onboard LED
- Set port direction
  - `echo -n "out" > /sys/class/gpio/gpio3/direction`
- Read/write value
  - `echo -n "1" > /sys/class/gpio/gpio3/value`
  - `echo -n "0" > /sys/class/gpio/gpio3/value`

Working with sensor data – ADC read

- For this example we'll use the Grove Shield with the light sensor connected to A0



Working with sensor data – ADC read

- First, set a multiplexer value to connect the GPIO to the ADC
  - `echo -n "37" > /sys/class/gpio/export`
  - `echo -n "out" > /sys/class/gpio/gpio37/direction`
  - `echo -n "0" > /sys/class/gpio/gpio37/value`
- Next, read directly from sysfs
  - `cat /sys/bus/iio/devices/iio\:device0/in_voltage0_raw`
- The Galileo's ADC chip (AD7298) can be temperature compensated for more accurate measurements
- Once you have exported the GPIO pins you need, you don't need to do it again

Back to node – fs and galileo-io

- There is a node module called *fs* to handle filesystem interactions

  -
    ```
    fs.readFile('/etc/passwd', function (err, data) {
      if (err) throw err;
      console.log(data);
    });
    ```
- We could use this to handle all GPIO interactions, but there is a nice npm wrapper called *galileo-io* to make this a little cleaner
  - This is only capable of digital read/write and analog read/write from individual pins
  - Other useful Galileo hardware requires a bit more (UART, I2C, SPI, etc)

Writing a server, part 4: data

- This example will stream new ADC measurements using socket.io
- A static content folder is needed to serve up the client-side JS
- server_data.js:

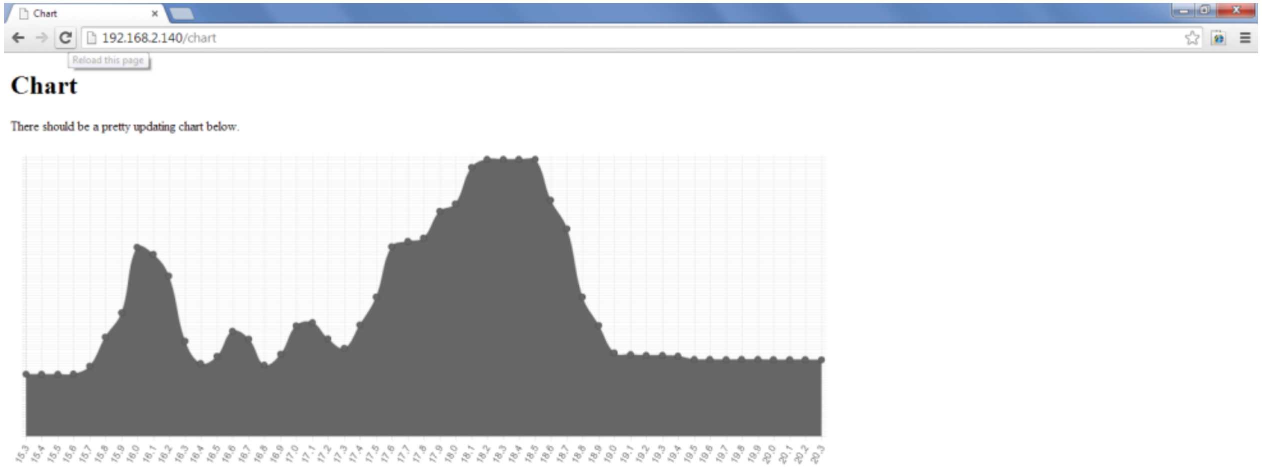
```
var Galileo = require('galileo-io');
var board = new Galileo();

app.use(express.static(__dirname + '/js'));
...
board.analogRead("A0", function(data) {
    io.emit('data', data);
});
```

- views/data.ejs:

```
<script src="jquery-1.11.1.min.js"></script>
<script>
    socket.on('data', function(msg) {
        $('#data').text(msg);
    });
</script>
...
<div id="data"></div>
```

Writing a server, part 5: chart

- Chart.js is a library to easily generate nice-looking plots
  - Other great visualization options in d3.js (Data-Driven Documents)
- server_chart.js includes a new route (‘/chart’) to utilize a new view (views/chart.ejs) to demonstrate this



Chart

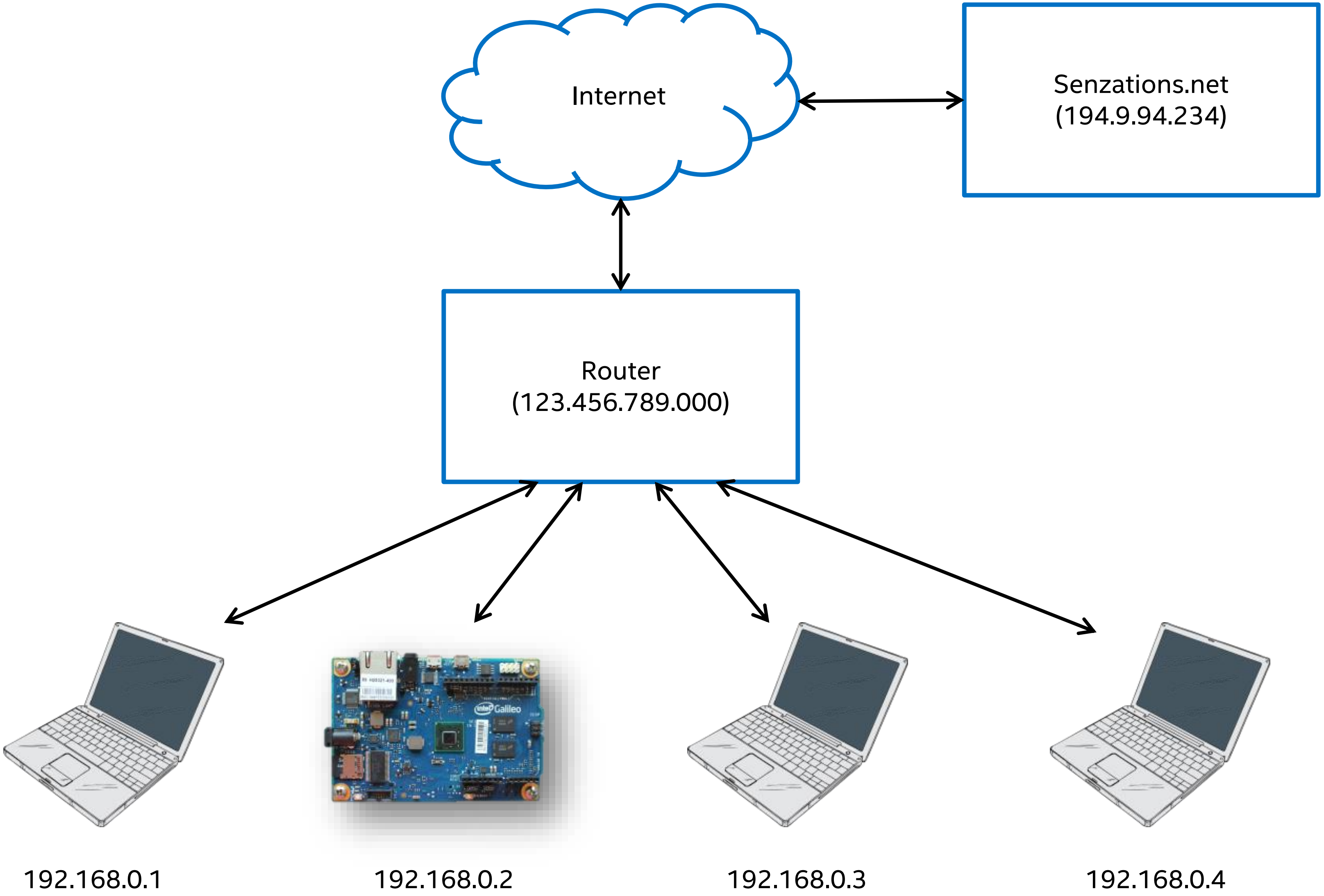
There should be a pretty updating chart below.

Download these slides (and examples)

https://github.com/jpwright/senzations14-galileo-nodejs

Questions

TCP/IP Network structure



Internet

Senzations.net
(194.9.94.234)

Router
(123.456.789.000)

192.168.0.1

192.168.0.2

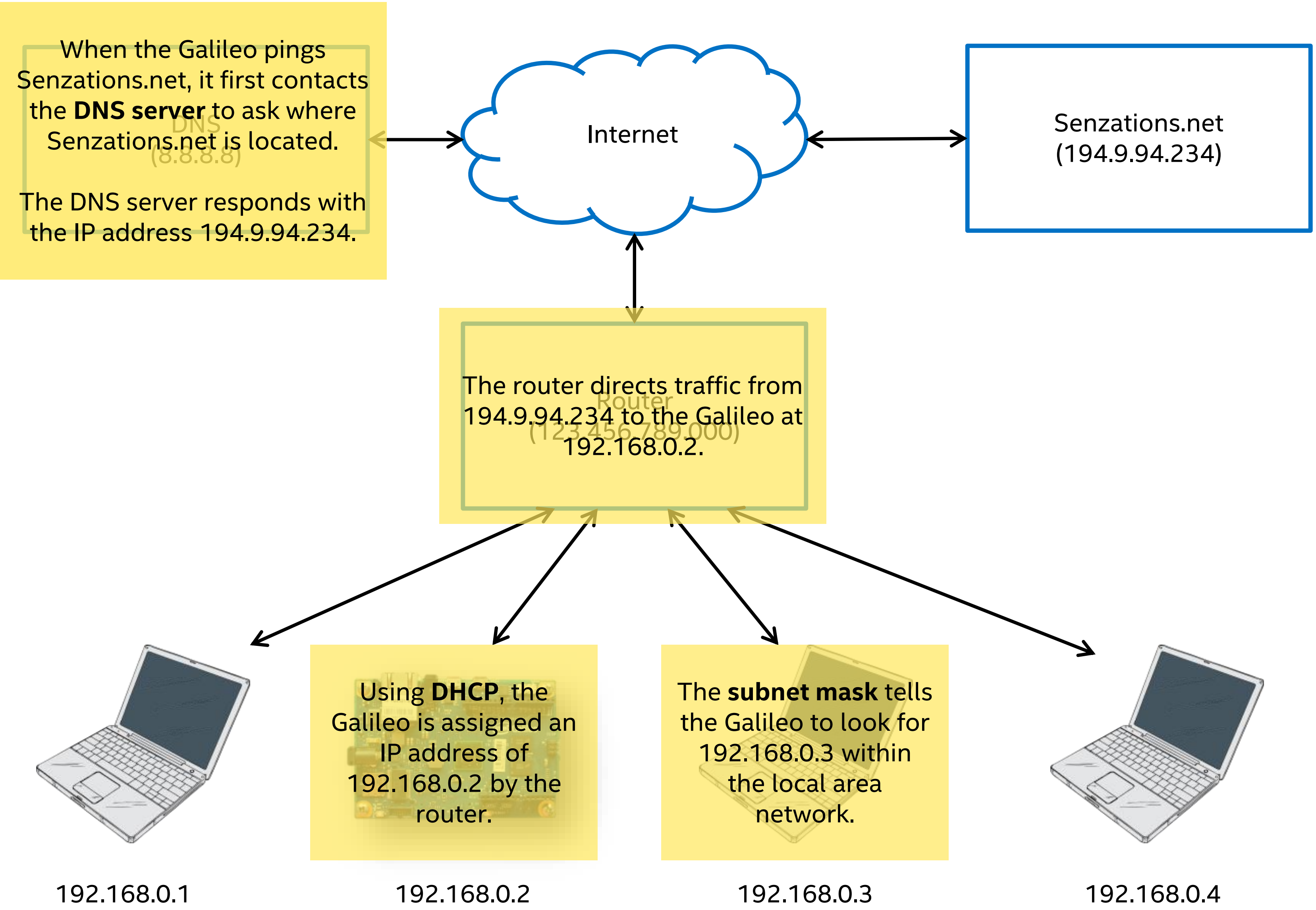
192.168.0.3

192.168.0.4

TCP/IP Network terminology

- **IP address** – unique designator for a device on a network
  - IPv4: 50.131.197.209
  - IPv6: 2001:0db8:85a3:0000:0000:8a2e:0370:7334
- **Local IP** – designator for a device on a local area network
- **External IP** – designator for a device to the entire Internet
- **DHCP** (dynamic host configuration protocol) – means for a router to automatically assign IP addresses to devices on its network
- **Subnet mask** – used to define a network prefix (e.g., use 192.168.0.x to refer to devices on the LAN)
- **DNS** (domain name service) – translate human-readable URL to an IP address of a server

TCP/IP Network structure



When the Galileo pings Senzations.net, it first contacts the **DNS server** to ask where Senzations.net is located.

The DNS server responds with the IP address 194.9.94.234.

The router directs traffic from 194.9.94.234 to the Galileo at 192.168.0.2.

Using **DHCP**, the Galileo is assigned an IP address of 192.168.0.2 by the router.

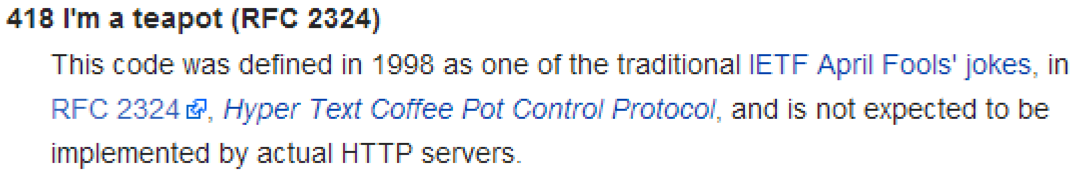
The **subnet mask** tells the Galileo to look for 192.168.0.3 within the local area network.

How do we run our own server?

- A server is just a device that responds to requests on a network, and which is typically always on
- Traffic of incoming/outgoing requests is divided into **ports** (the TCP part of TCP/IP)
  - **HTTP** (web) – port 80
  - **SSH** (secure shell) – port 22
  - **FTP** (file transfer protocol) – port 20
  - **SMTP** (mail) – port 25
  - Anything else you want
- Ports are typically appended on to the end of an address, e.g. 192.168.0.1*:80*

How do we run our own server?

- An **HTTP server** typically listens on port 80 and responds with **HTML content** designed to be viewed in a web browser.
- The HTTP protocol has standard **headers** in each request
  - **User-Agent**: the type of device making the request
  - **Content-Type**: how the body of the request is formatted
  - **Referrer**: the previous page from which a link was followed
  - Many more
- The HTTP server will respond with one of many **status codes**
  - **200**: Everything's OK
  - **401:** Not authorized
  - **404**: File not found
  - **418:** I'm a teapot



418 I'm a teapot (RFC 2324)

This code was defined in 1998 as one of the traditional IETF April Fools' jokes, in RFC 2324, *Hyper Text Coffee Pot Control Protocol*, and is not expected to be implemented by actual HTTP servers.