Lecture 2

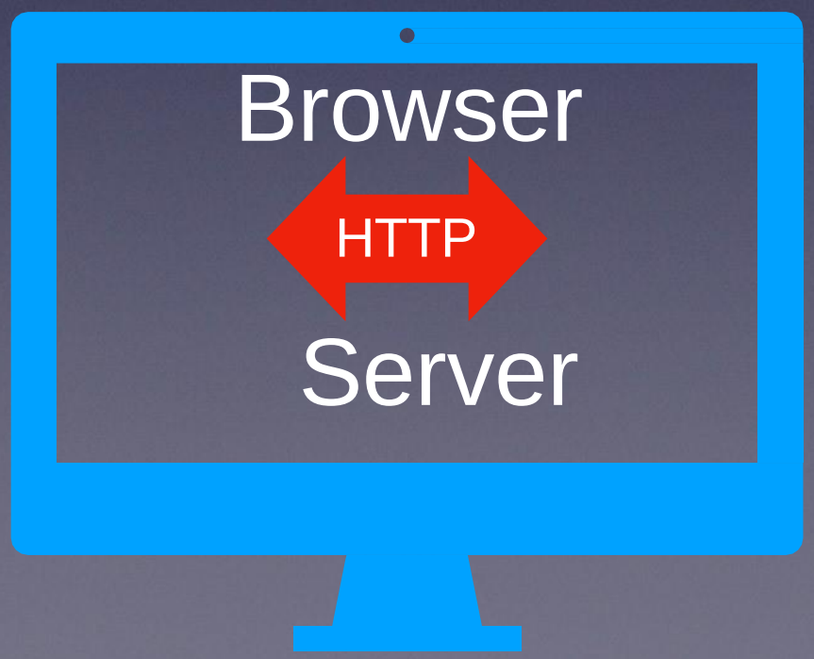
Cloud services and HTTP

This lecture covers:

- Last lab
- Cloud services
- URIs and HTTP protocol
- A new lab

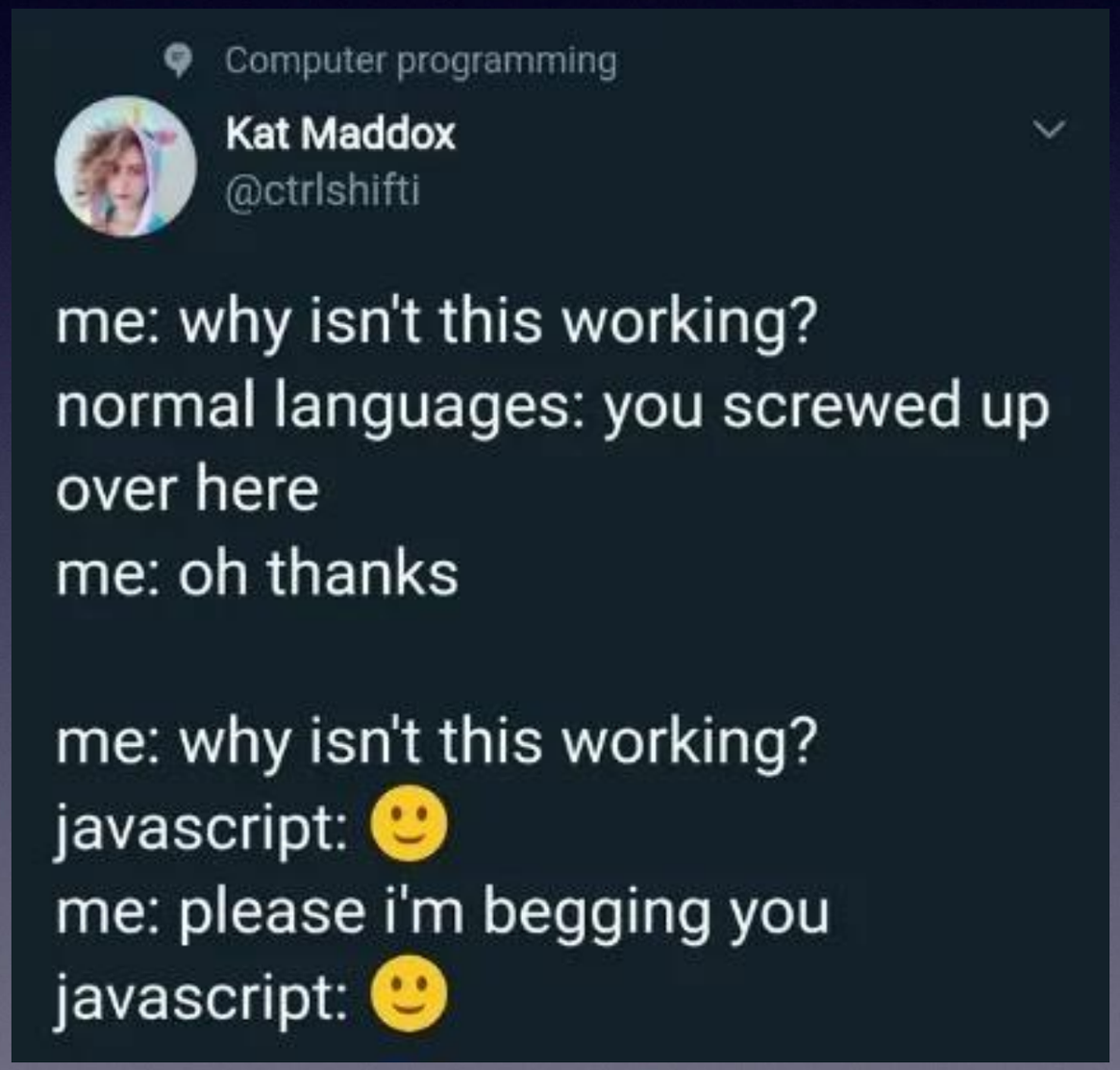
Last lab

- You had a server and a client working
- You hosted it locally on just 1 computer
- You "remembered" HTML, CSS, JavaScript, Node.js, Express, AJAX, jquery…



Hopefully….

You also got all the fun stuff about debugging with JavaScript

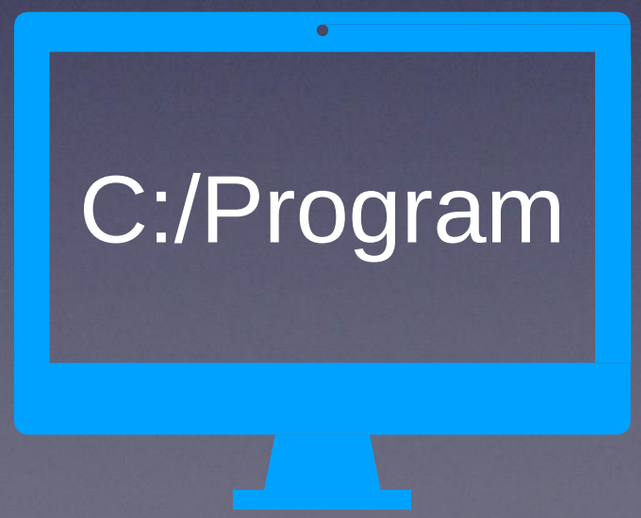


Path names?
(Why are RGU computers so weird?)

- Separation of Program space and Storage space

- You have an H:/ drive (and a OneDrive) for storage

- The computer runs stuff from the C:/ drive



Damn you firewall!!

Like a network within your local machine.

If server and browser are both on the same machine, the firewall is not breeched.
So, cancel it away every time.

But you couldn’t access the website from the machine next to yours.

Damn VIM!!

If you’re using git or gh on the command line:

git commit –a –m “Here is a nice comment”

But use your own comment. Otherwise it throws you into VIM and it is hard to escape.

:qa

In PowerShell (Or VS's Terminal)

To start recognising things, switch to the H:/ drive using:

cd H:/

Then use ls and cd and the tab key to find where your stuff is.

What it’s all about

- Up key
- Tab key
- Path names

Am I behind already??

Basic Aims of Last Lab:

- Use PowerShell (or some command line)
- Create and use GitHub repos
- Create a back-end, front-end and serve them locally

Last Lab

GitHub outside RGU

GitHub

git push

You in RGU

What goes in GitHub?

- Files that will change (or that are required by your program)
- Files that cannot be automatically generated

What does **NOT** go in GitHub?

- Files that can be automatically generated (e.g. node_modules, binary files)
- Anything that should be private (GitHub token, usernames and passwords, urls that contain usernames and passwords)

What is a cloud service?

Two main options for hosting web systems:

- Your very own server which you buy and connect to the internet on a static IP address
- Someone else’s computer (or space on someone else’s computer), already has a static IP address

Which is better?

Your very own server

Advantages

- You have direct access to your server
- Complete control
- Costs?

Your very own server

Disadvantages

- You are responsible for your server. What if it goes down?
- What if it isn’t big enough? Or small enough?
- Costs?

Server Specs

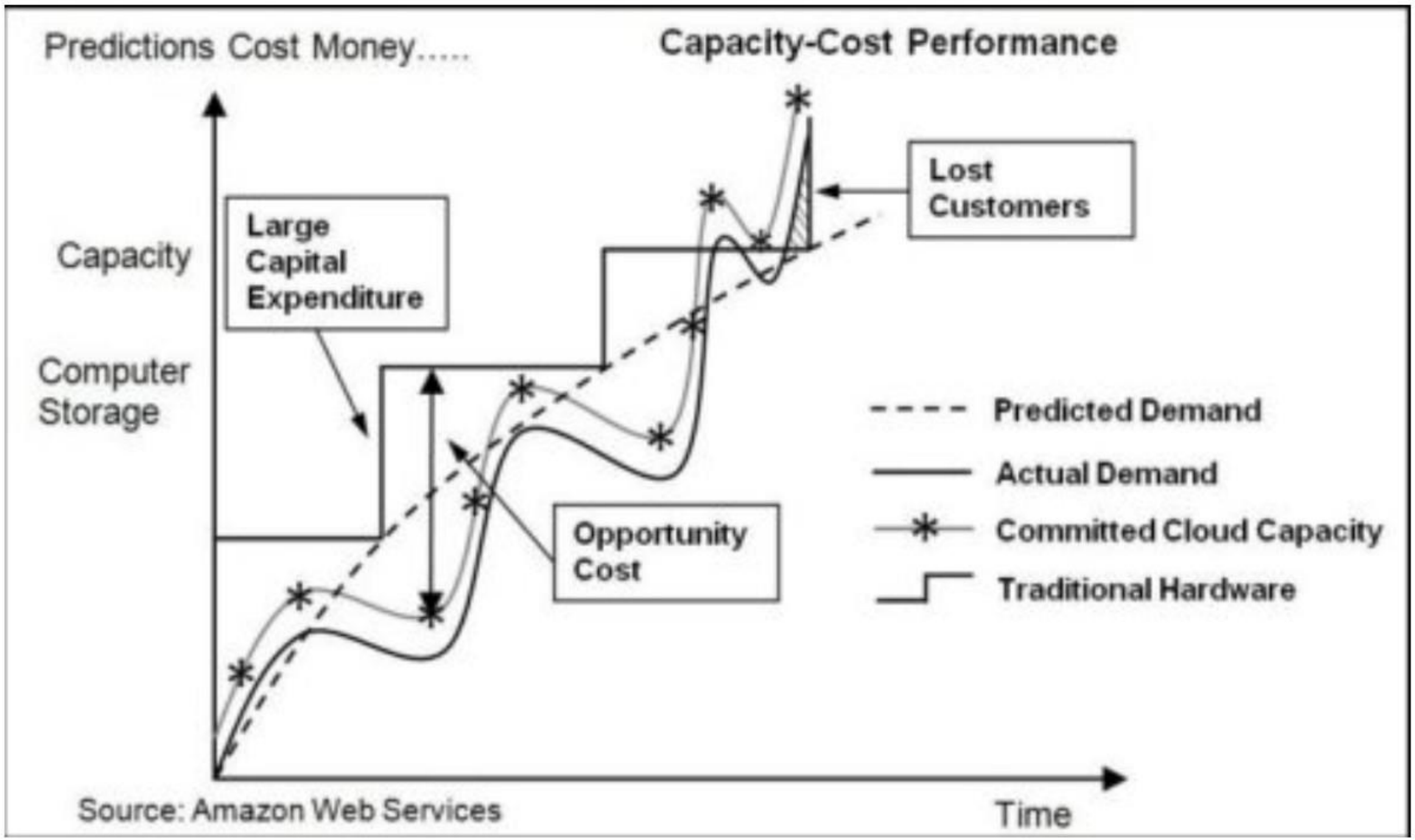
Your web system needs a certain specification to run. Just as you might choose a computer with a specific CPU, RAM, storage… you can also select cloud instances with specific CPU, RAM, storage…

Think about costs:

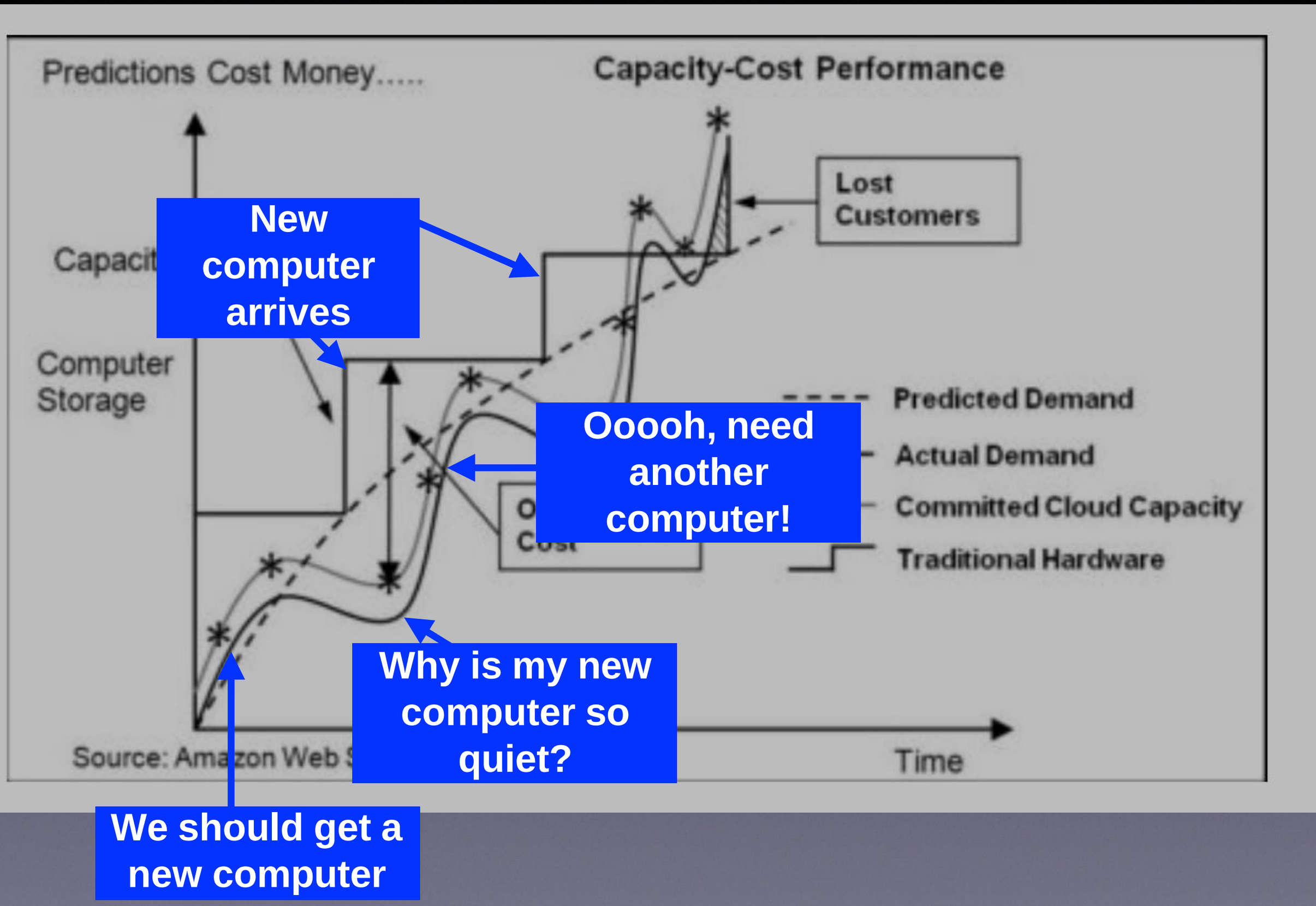
https://www.heroku.com/pricing

https://calculator.aws/

Capacity vs Utilisation



Capacity vs Utilisation



Cloud

Using a cloud instance is:

- Flexible
- “Easier”
- More future proof
- **Not** necessarily cheaper!

…as a service

Traditionally 3 main models of cloud provision:

- Infrastructure-as-a-Service (IaaS)
- Platform-as-a-Service (PaaS)
- Software-as-a-Service (SaaS)

Services: IT resources bundled up and offered to consumer

But we now also have DBaaS (for databases), BaaS (back-end), AIaaS (artificial intelligence)….

Infrastructure as a Service

- offer infrastructure-centric IT resources: e.g. virtual machines, network, connectivity, operating systems, other “raw” IT resources
- high level of control & responsibility over configuration & utilisation
- resources usually not pre-configured, cloud consumer must configure them before using, and maintained e.g. after creating a virtual server, you might need to install the OS, then the server software, etc.
- Examples:
  - Digital Ocean
  - Microsoft Azure
  - AWS EC2 instances and storage options
  - Rackspace
  - IBM

IaaS

- YOU decide what is installed on it (and it could just be Docker…)
- YOU have responsibility for upgrading things (but they could be automatically scheduled)
- What happens if it dies?

Platform as a Service

- a pre-defined, **ready-to-use environment** of already deployed and configured IT resources for development
- users can create and deploy applications using supported programming languages and tools
- applications to be built on top of existing platform
- no management of underlying cloud infrastructure needed - platform is usually presented as an API; details of platform are hidden from service consumers
- Examples:
  - Google App Engine, AWS Lambda/Elastic beanstalk, Heroku, Vercel

PaaS

- You have to jump through some hoops to make things work (code in a specific format, use specific packages)
- An upgrade from the service provider might break your work
- But it will still run code

Software as a Service

- No maintenance, no development, just use!
- A software program as a shared cloud service
- Users access service using a thin client (e.g. web browser)
- Accessible from different platforms
- Limited administrative control by users
- A user can only configure his/her own settings
- Examples: Gmail, Google Docs, Office 365, Adobe Creative Cloud, GitHub

SaaS

- Easy to use
- No responsibility for upgrades
- Lesser degree of flexibility (interoperation)

What is Moodle?

We create online learning resources using Moodle, it’s a “web technology” but it could be:

A. Infrastructure as a Service

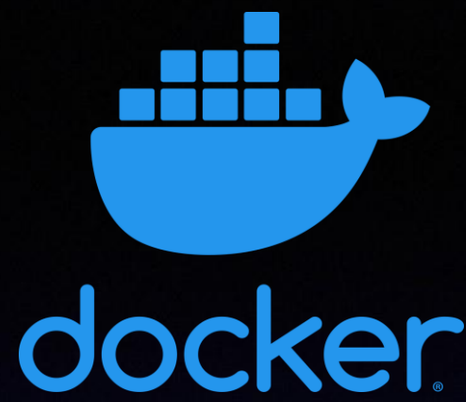
B. Platform as a Service

C. Software as a Service

Moodle could be Software as a Service

- You can put stuff online
- It’s customisable to an extent
- You have limited control over the features

Aside about Docker



- It’s a series of PaaS tools. A fast way to turn IaaS into PaaS.
- It is like a whole computer in a file.
- You can get Docker images from DockerHub and use them to spin up your own containers.
- Similar to a virtual machine but docker is smaller, faster and can share drives with the host machine.

URLs, URIs and Ports

Some services have specific port numbers associated with them:

- HTTP -> 80
- SSH -> 22
- HTTPS -> 443
- MongoDB -> 27017

Sometimes (especially when running a server locally), you'll use a port number.

URI

Uniform **R**esource **I**dentifier

String of characters that unambiguously identifies a resource. URL (**U**niform **R**esource **L**ocator) is a sub set of this.

Is defined by RFC 3986 - an internet standard.

Wait…RFC?

- RFC stands for “Request For Comments” and is a “technical standard” from the Internet Society
- It is often how we decide how to position 1s and 0s so that other computers on the internet know what we’re saying

URI Syntax

https://www.rgu.ac.uk/study/course-search?school_filter=SOC

scheme

path

https://www.rgu.ac.uk/study/course-search?school_filter=SOC

authority

query

Examples from the lab

file:///Users/pam/ENISS/simpleWeb/index.html

- Accessing a file on **my** local computer
- **/Users/pam/ENISS/simpleWeb/** - this is the path
- The file is called **index.html**

Examples from the lab

http://127.0.0.1:8000/

- A server running on **my** local computer
- http:// is the protocol
- 127.0.0.1 is the “loopback” address (it means “this computer” or localhost)
- 8000 is the port number

Examples from the lab

http://soc-cm4025-01:8000/myWebpage.html

- My page hosted on a computer on the local network
  - http:// protocol
  - Local computer name
  - Port is 8000 (but 80 is the default for http)

What about IP addresses?

DNS = Domain Name System

It is a server that turns human-readable names into numeric IP addresses.

Create your own URI

- Protocol
- Address (including ports if necessary)
- Query (if appropriate)
- This takes us in to RESTful APIs….

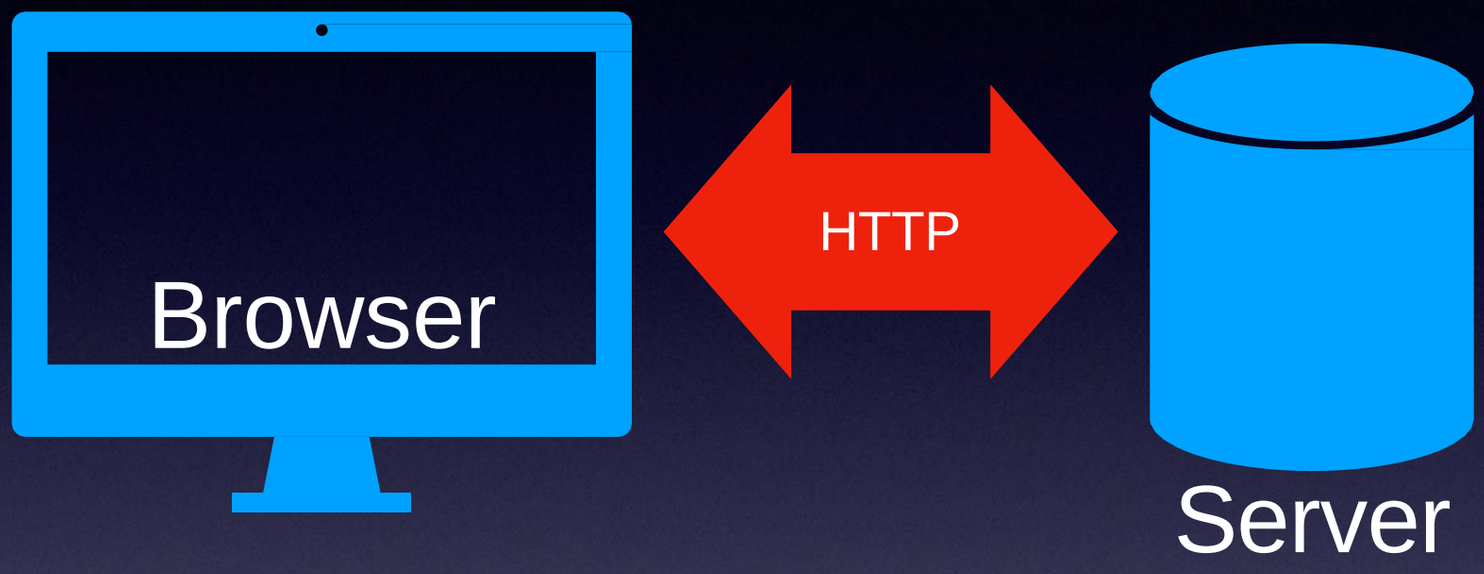
HTTP

HTTP is a protocol defined in RFC 7230 (and others referred to therein).

Browsers from the address bar use HTTP “GET” requests to retrieve data from a URI.

There are other options (Postman, curl).

HTTP



- Browsers send HTTP requests
- Servers respond in HTTP

HTTP Methods

Method	Description	Sec.
GET	Transfer a current representation of the target resource.	4.3.1
HEAD	Same as GET, but only transfer the status line and header section.	4.3.2
POST	Perform resource-specific processing on the request payload.	4.3.3
PUT	Replace all current representations of the target resource with the request payload.	4.3.4
DELETE	Remove all current representations of the target resource.	4.3.5
CONNECT	Establish a tunnel to the server identified by the target resource.	4.3.6
OPTIONS	Describe the communication options for the target resource.	4.3.7
TRACE	Perform a message loop-back test along the path to the target resource.	4.3.8

From RFC 7231

HTTP and Servers

- Servers respond to HTTP requests
- They **must** respond to GET and HEAD but all the others are optional
- They respond with a **response code**

HTTP response codes

Code	Reason-Phrase	Defined in...
100	Continue	Section 6.2.1
101	Switching Protocols	Section 6.2.2
200	OK	Section 6.3.1
201	Created	Section 6.3.2
202	Accepted	Section 6.3.3
203	Non-Authoritative Information	Section 6.3.4
204	No Content	Section 6.3.5
.....		
400	Bad Request	Section 6.5.1
401	Unauthorized	Section 3.1 of [RFC7235]
402	Payment Required	Section 6.5.2
403	Forbidden	Section 6.5.3
404	Not Found	Section 6.5.4
405	Method Not Allowed	Section 6.5.5
406	Not Acceptable	Section 6.5.6
.....		
500	Internal Server Error	Section 6.6.1
501	Not Implemented	Section 6.6.2
502	Bad Gateway	Section 6.6.3
503	Service Unavailable	Section 6.6.4
504	Gateway Timeout	Section 6.6.5
505	HTTP Version Not Supported	Section 6.6.6

From RFC 7231 - but you can get them from other sources

How can I see these codes?

- Pick a browser
- Load up the developer tools (e.g. in Chrome it’s View->Developer->Developer tools)
- Look at the network tab to see all the network traffic that goes into making a web page

I- 4 C @ 127.0.0.1.8000 j/ @ update

Pam's Second Amazing Bootstrap Page

Resize this responsive page to see the effect!

Column 1

Totes amazeballs

Guide to AWS ed....pdf

g Elements Console Sources Network Performance Memory Application Security Lighthouse

@ @ @ Preserve log Disable cache Online v V X

filter Hide data URLs XHR JS CSS Img Media Font Doc WS Manifest Other Has blocked cookies

Blocked Requests

20 ms 40 ms 60 ms 80 ms 100 ms 120 ms 140 ms 160 ms 180 ms 200 ms

	Status	Type	Initiator	Size	Time	Waterfall
127.0.0.1	200	document	Other	1.1 kB	20 ms	
bootstrap.min.css	200	stylesheet	(index)	(disk cache)	2 ms	
jquery.min.js	200	script	(index)	(disk cache)	5 ms	
bootstrap.min.js	200	script	(index)	(disk cache)	4 ms	
favicon.ico	200	text/html	Other	1.1 kB	3 ms	

5 requests 2.3 kB transferred 253 kB resources Finish: 204 ms DOMContentLoaded: 188 ms Load: 178 ms

; Console

@ top Filter Default levels v

Show all X

It's the developer tools Network tab, but doesn't it look ugly in this slide?

Lab 2

- You're getting a virtual Linux box to play even if you didn't ask for one!
- The folders on it are way more simple than on Windows.
- You *can* use GUI tools, but the lab only considers a terminal
  Don't worry if you don't get on to Docker.

Pam will stop here if time is short. Understanding RESTful might be useful in the future, but is not vital for your assessment.

Where does REST fit in?

- A RESTful API breaks down a complicated procedure into little modules
- It is a means of designing an API - a software architecture
- It’s stateless
- It takes advantage of HTTP:
  - GET
  - PUT
  - POST
  - DELETE

REST

- **Re**presentational **S**tate **T**ransfer
- It is a **software architecture** - a way to design a back end API so that it can be easily understood by the people who might use it.

HTTP Methodologies

- **GET**: retrieves a resource (from the server)
- **PUT/PATCH**: replaces/updates an existing resource (on the server)
- **POST**: creates a new resource (on the server)
- **DELETE**: destroys the resource (on the server)

If I want to:

- **Create** a user on my web site: **POST**
- **Change** a user’s details: **PATCH** (or **PUT**)
- **Retrieve** an html form: **GET**
- **Delete** a user record: **DELETE**

What is CRUD

- Stands for **Create, Read, Update, Delete**
- One-to-one mapping with **POST, GET, PUT, DELETE**
- Tells you how to design a RESTful API

In a RESTful API

A single URL defines a resource. That means that the different HTTP methods of accessing it define what should happen to the resource.

In a RESTful API

The **Client** and **Server** are loosely coupled and treated independently. Which ties in nicely with some other design patterns.

In a RESTful API

All client-server operations are stateless. Any state management should take place on the client.

That means that the server does not **remember** the client between transactions

In a RESTful API

Intermediate systems like load balancers or proxies should not affect communication between client and server

In a RESTful API

The server can supply **code on demand** to the client to increase their functionality.

That means JavaScript (or Java applets)

RESTful example

My resource is a user database, and its url is:

http://api.example.com/users

I add a user by calling:

POST http://api.example.com/users?username=Pam

RESTful returns

- A RESTful API also considers the HTTP responses

- Returning the correct HTTP code can give more details to the client

- e.g. returning code 201 on successfully adding a user; 404 for not found; 400 for bad request

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

REST

- Resources are URIs are nouns (e.g. username, item)
- HTTP methods are verbs (e.g. add, delete, fetch)

Now…

- On to the lab
- You're getting a virtual Linux box to play with
- The folders on it are way more simple
- You *can* use GUI tools, but the lab only considers a terminal
- Don't worry if you don't get on to "Docker"

A word on Docker

This is optional. You can complete this week without it. But it is the current buzz word, so it might be good to have a go at setting it up and getting it working.