



Trinity Saint David

(Gweler yr ochr arall am y fersiwn Cymraeg)

1. a). Navigating to `https://google.com`:

- Browser must use dns to find IP of site. ✓
- Check host file.
- If not in host file, query dns server
either find authoritative server for ~~the~~ recursive query
or non-authoritative for iterative.

Recursive query ~~is that~~ asks the authority and loops through a bunch of servers to try to dns. Returns "yes" or "no".

Iterative query asks non-authority and can be forwarded so it returns "yes" or "try here" and redirects you.

~~Once dns is used to find IP address of google.com,~~

- Query `.com` ~~first~~ dns server first.
then google server.
- You'll end up with the IP for google.com.
- Browser sends a GET HTTP request to google server but needs to initiate session first.
- TCP session established with google server.
- TCP/IP packet sent.
- HTTP sends verbs.
- Google server ~~receives~~ receives HTTP verbs and returns responses probably in format of response codes eg 200 OK.

- | | | |
|-----------|-----------------|-------------------------|
| app | → data + header | → packets encapsulated. |
| present | ↓ | |
| session | ↓ | |
| transport | ↓ | |
| network | ↓ | |
| data link | ↓ | |
| phys | ↓ | |

+ header
+ header

majorly TCP/IP packet has payload, address, mac, trailer.

1. b). URN = ^{Uniform} universal resource name.
has name but no ~~location~~.
eg ISBN : - - - -

~~URL~~ URL = universal resource ^{location} link. ✗
has ~~name~~ and ~~location~~

eg `stuff.home.mysite.com/givememypic.jpg`

URI = universal resource identifier
~~the~~ is URNs and ~~the~~ URLs.

2. a).
- GET - return a resource
 - POST - send a new resource
 - PUT - update a resource
 - DELETE - delete a resource

6

2. b). Idempotent → you can do it more than once and every time the result will be the same.

A GET request is idempotent but a POST isn't ~~because~~ because you're changing the database when you add something.

PUT should be idempotent because you update the same thing with the same changes each time.

DELETE isn't idempotent because you're deleting stuff.

4

2. c).	200 - OK with body	100	continue
	401 - unauthorised	200	OK
	403 - forbidden	204	nothing to return
	404 - not found	400	bad request
	500 - server error	401	unauthorised
		403	forbidden
		404	not found
		500	server error
		503	service unavailable
		504	session timeout

200 = request was fine and stuff was returned.

401 = you don't have authorisation, maybe not logged in.

403 = you're not allowed to see this.

404 = can't find this resource.

500 = generic server error.

10

2. d). 401 - unauthorised → you could see it if you were logged in and had correct authorisation.

403 - forbidden → you should never see it. You won't get permission if you log in.

3

3. a). REST :

Universal framework for building web services.

Uses HTTP verbs to communicate.

~~Rest~~ Built with dynamic ~~type~~ resource links instead of hard-coded URLs.

Resource-oriented so you interact with ~~resources~~ instead of ~~the act~~ resource representations instead of the actual data.

Benefits — aims to provide one way for web services to communicate ~~so~~ rather than a bunch of different ways.

This will improve compatibility.

Should improve scalability.

Improves connectedness of services because they'll be built around hyperlinks (HATEOAS) so it should be easier to tell the possible paths through a service.

Hard-coded links must be changed when the resource changes but REST means that the link should still ~~be~~ work when resources are altered.

Stateless — don't worry about ~~sessions~~ in this layer at least.

Communicates as HTTP verbs ↔ response codes — simple, standard.

8

3. b). JPA & access to database.

~~send query.~~

- Set up resource representation classes, matching db tables.
- Controller to set up paths.
- Also set up database ~~(links)~~.
- In entity classes, use annotations to tell which field is Id * and to set up ~~links~~ and foreign keys between tables.
- Marshall the JSON so you get the correct format when you query.
- Set up GET, PUT, POST and DELETE paths.
- Use postman to send queries and it should return data from database.
- Add list classes for returning the list of objects in a table eg /batch/2/listallplates should return the list of all plate objects in batch number 2.

*

~~or Hibernate~~

- Repository layer is pretty good for abstraction and accessing db.
- Hibernate has unit-of-work so * whole transaction succeeds or fails, not bits.
- ORM means you can map your db to your classes which is easier but less refined.
- ~~I guess a mismatch is bad.~~
- Use application/json for queries or maybe application/xml, as long as you set it up in the controller.

12

4. a) Cloud ☒

Good — you don't need a server room (space).

- you don't need to maintain servers.
- you don't need a whole server when you only use some of the space.
- you can pay less to use less.
- you can increase your server space to scale up.
- your service can be split up across multiple servers.
- you can scale part of your service independently.
- your service probably won't go down all at once if something goes wrong in one place.
- you don't need server architects.
- you don't need to tailor your service to your architecture or vice versa.
- service can be available world wide and in less economic areas

Bad — your information is on someone else's machine.

- you can't fine tune your service as much.
- you don't have the option to ~~not~~ not gather user data because the servers do that and you don't own the servers.
- a server in USA follows USA laws, not EU laws.
- you can't fix broken stuff — it's not your job.
- if they raise prices, you don't have servers or architects so you'll probably pay.

4. b). Vulnerabilities — Security (injection, logins)
- one service going down
 - scaling

Must consider security as in how logins will work

Basic — login system, not secure, must use HTTPS to secure

The other one — hashed login

OAuth — API Keys and secret user keys

Certificates — Certificates as proof of agreement.

Some require HTTPS.

Must consider security as in whether the system is vulnerable to SQL injection or scripting etc. If it is vulnerable, it needs to be patched.

Must consider what happens when part of the service becomes unresponsive. Circuit breaker pattern might help.

Must consider how scaling might happen in future. You don't want to lock yourself into a small, unscalable service. It's better to build it with separation so individual service scaling is possible.

S



4. c). Automation:

Good — You get extra tests you didn't need to write.

- You can get an idea of how your client will look and work.
- You don't need UML as much or at all.
- Your client should match your service.
- You don't need to code as much from scratch.
- You can see/use the response from Swagger to tailor your events and commands.
- You can make new endpoints in the same style as your existing ones.
- Should be less work but might not be.
- Good for checking outputs and endpoints.

6