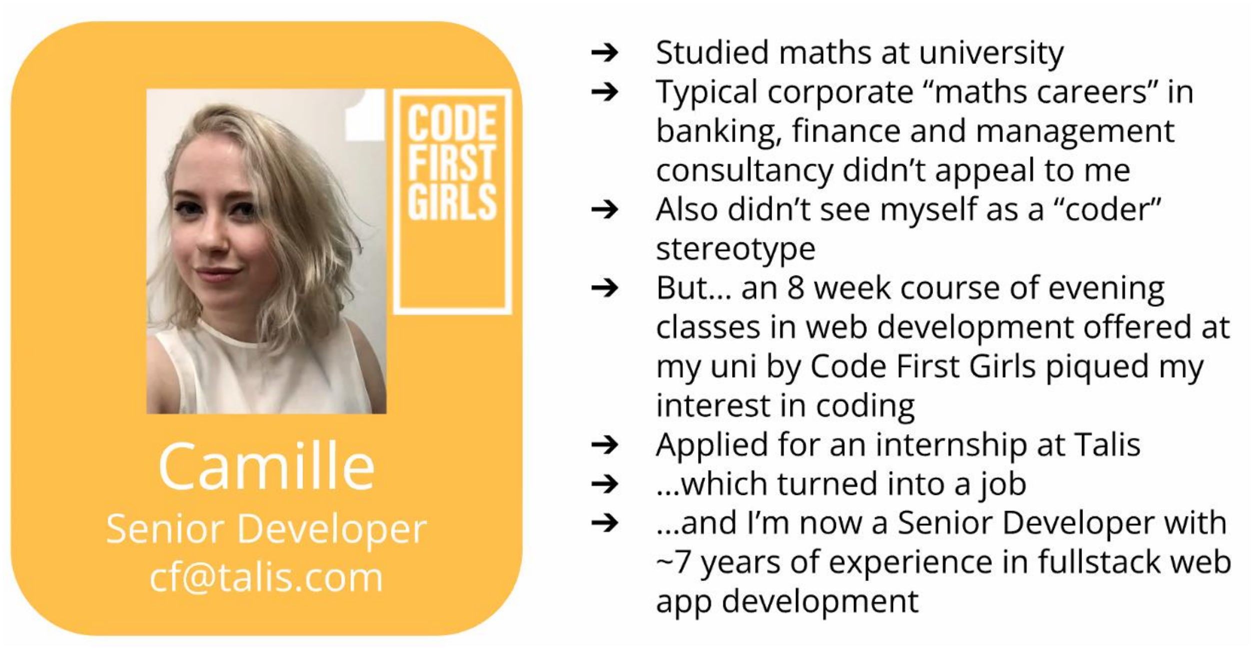
Nadeem Shabir & Camille Fenton

16/6/22

Introduction to RESTful APIs

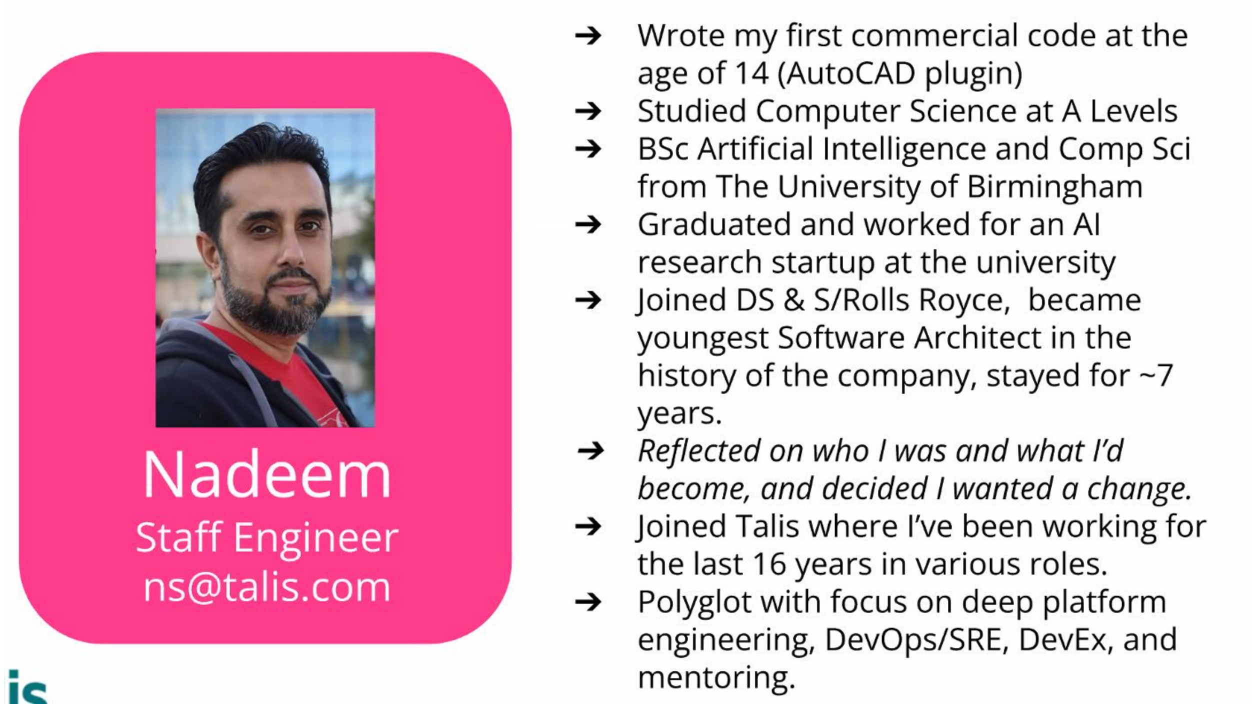
About the speakers:

Both work at Talis.


CODE FIRST GIRLS

Camille
Senior Developer
cf@talis.com

- ➔ Studied maths at university
- ➔ Typical corporate "maths careers" in banking, finance and management consultancy didn't appeal to me
- ➔ Also didn't see myself as a "coder" stereotype
- ➔ But… an 8 week course of evening classes in web development offered at my uni by Code First Girls piqued my interest in coding
- ➔ Applied for an internship at Talis
- ➔ …which turned into a job
- ➔ …and I'm now a Senior Developer with ~7 years of experience in fullstack web app development



Nadeem
Staff Engineer
ns@talis.com

- ➔ Wrote my first commercial code at the age of 14 (AutoCAD plugin)
- ➔ Studied Computer Science at A Levels
- ➔ BSc Artificial Intelligence and Comp Sci from The University of Birmingham
- ➔ Graduated and worked for an AI research startup at the university
- ➔ Joined DS & S/Rolls Royce, became youngest Software Architect in the history of the company, stayed for ~7 years.
- ➔ *Reflected on who I was and what I'd become, and decided I wanted a change.*
- ➔ Joined Talis where I've been working for the last 16 years in various roles.
- ➔ Polyglot with focus on deep platform engineering, DevOps/SRE, DevEx, and mentoring.

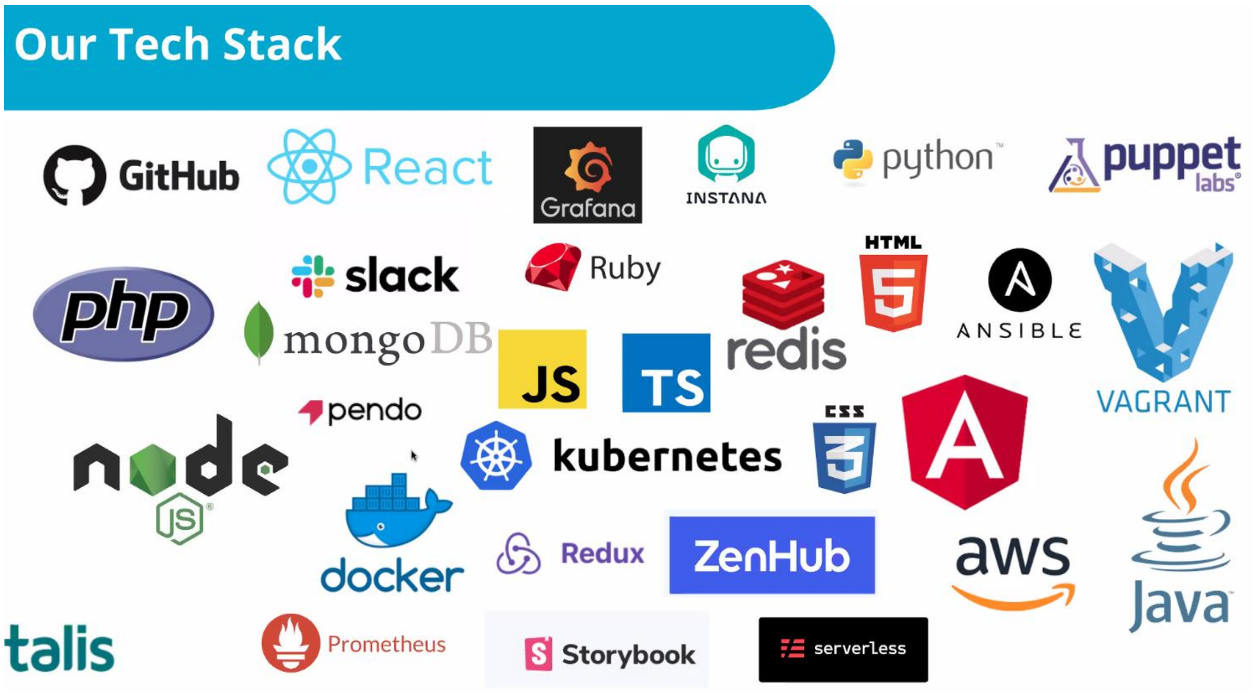
Background on Talis:

- Talis is an edtech software company
- Been around for over 40 years. Originally implementing library cataloguing systems for universities; now focussed on teaching and learning
- We build products, provide services and strive to apply technology to make education more connected

Used to be based in Birmingham but now:

- Fully remote company with majority of employees in the UK. Also have colleagues in Europe, Australia and North America
- Whole company is around 40 people
- Customers include over a hundred universities across 3 continents. Millions of users access their services every day
- Development team is roughly 20 people and they are looking to grow!

Diverse tech stack due to how long the company has been going:



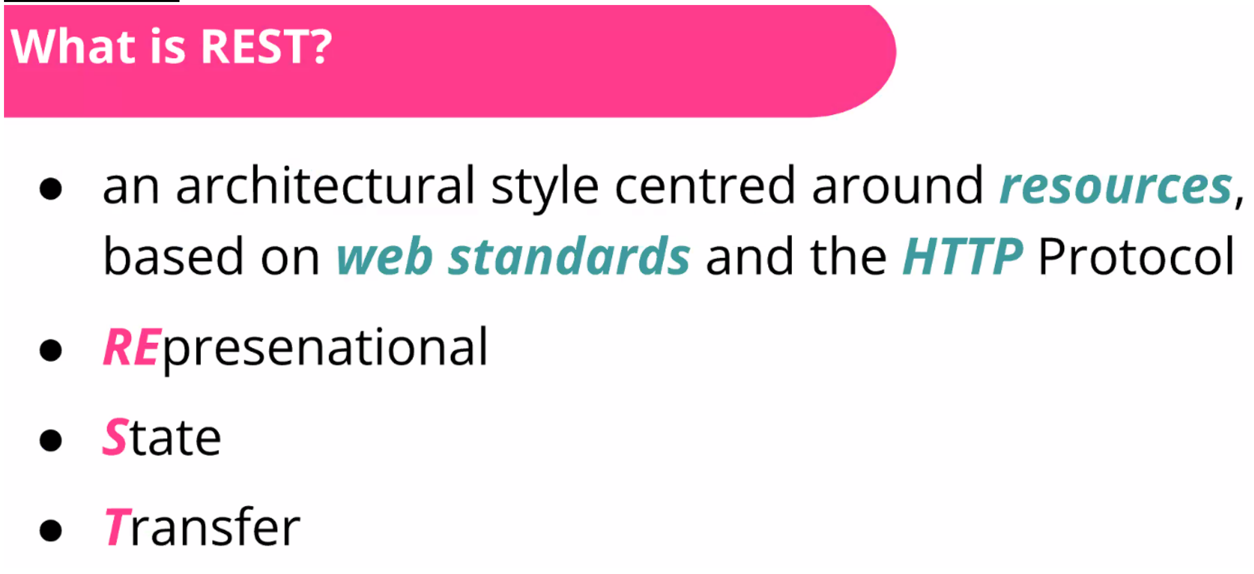
Our Tech Stack

Their philosophy is to try and use the technology that is best to solve the problem in front of you, rather than sticking with a specific language.

So, looking to hire developers who are willing to learn and change, but also unlearn and forget about stuff!

Camille adds this slide can look intimidating, don't be put off as not every dev at Talis is an expert at all of these! So not expected to know everything and tech is always changing so couldn't possibly know

<u>What is Rest:</u>



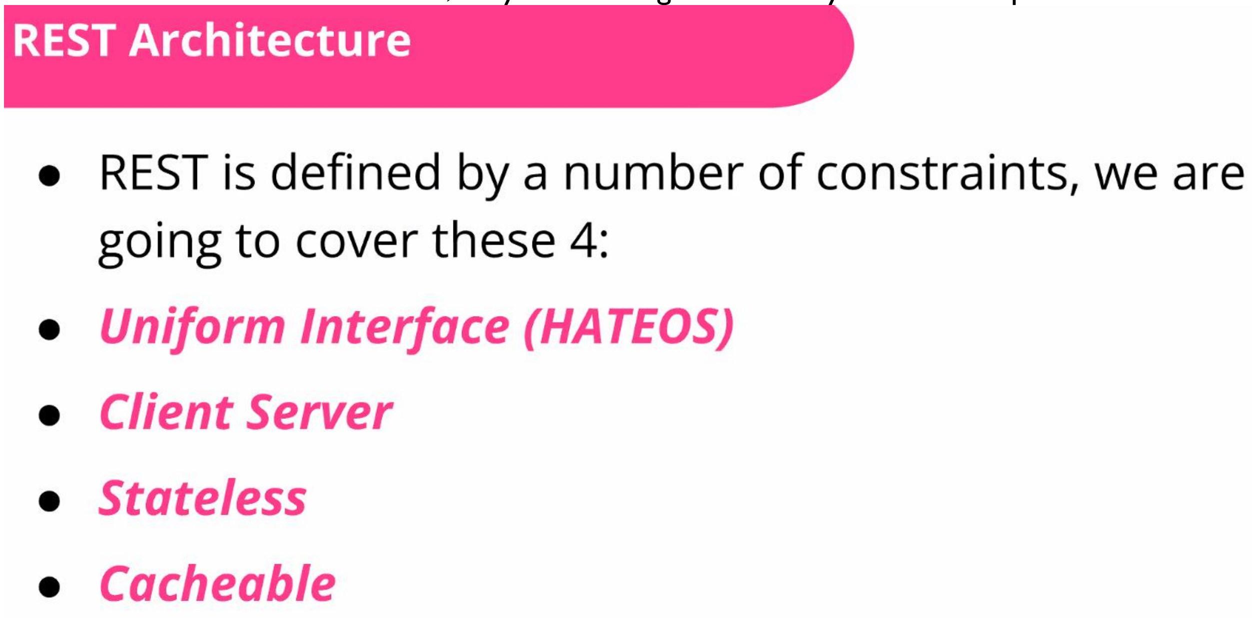
What is REST?

- an architectural style centred around ***resources***, based on ***web standards*** and the ***HTTP*** Protocol
- ***RE***presenational
- ***S***tate
- ***T***ransfer

Usually, diversity in the way we do things and make things is the best thing. But an API is a "contract" between you and your many clients – it needs to be standardised so it's accessible to everyone and they know how to use it!

Rest Architecture:

REST has more than 4 constraints, they are looking at what they feel most important



REST Architecture

- REST is defined by a number of constraints, we are going to cover these 4:
- ***Uniform Interface (HATEOS)***
- ***Client Server***
- ***Stateless***
- ***Cacheable***

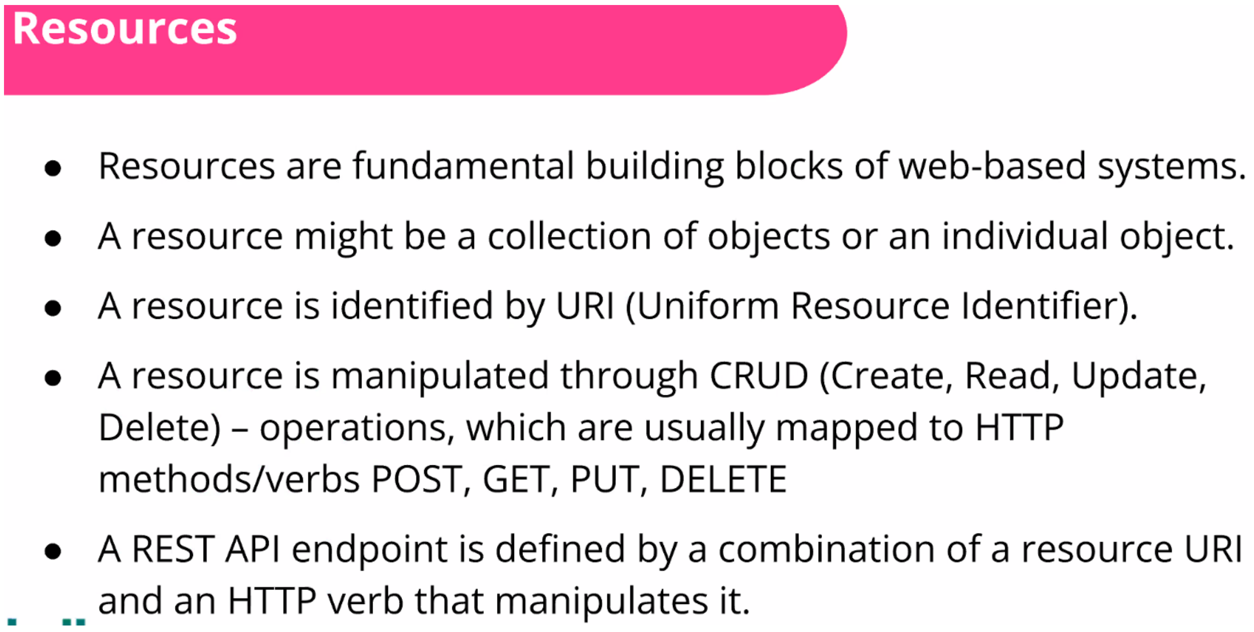
HATEOS – should be HATEOAS - Hypermedia as the engine of application state – means you have a uniform interface for interacting with your resources. So each resource is identified by a URI

Client/Server – separating the two. The client is not concerned with storing data, just concerned with what user is doing. Server deals with data storage.

Stateless – should not have knowledge of any previous request, so each request should be treated as "new".

Cacheable – When you make a request, you know every time you are going to get the same response to that request. So doesn't depend on a previous request. This means you can cache a response and it will always be valid. Caching means server doesn't have to run a full DB query every time, until a certain "constraint" changes (could be time limit, changes etc), for example. This is useful for scale – if you get a million requests to your server for the same webpage, you want to run the request once then cache the answer and provide the cached answer for the rest! Otherwise you are placing a heavy load on your server and it may slow down or break.

What are resources:

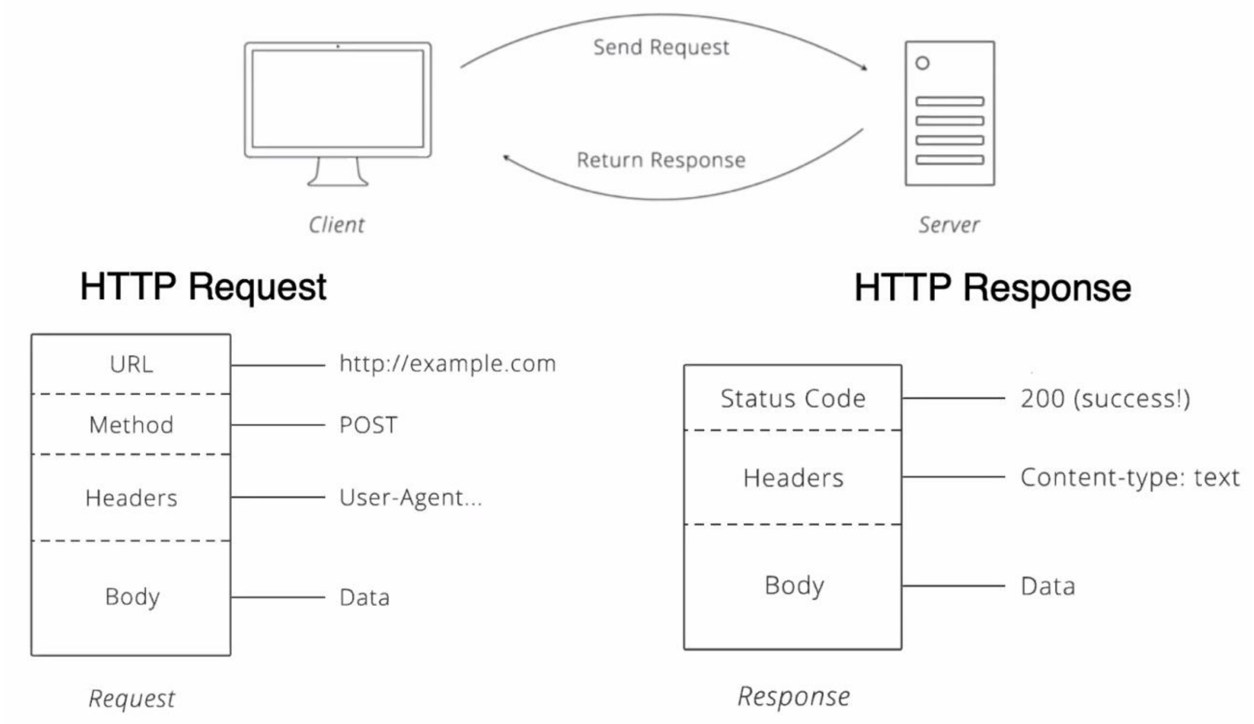


Resources

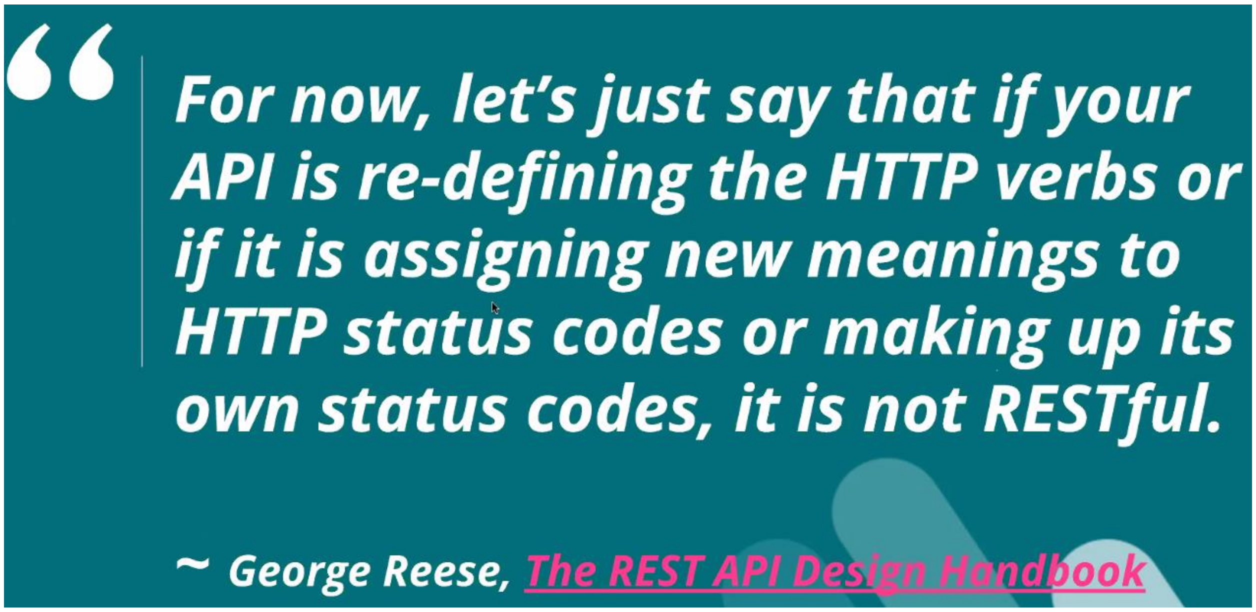
- Resources are fundamental building blocks of web-based systems.
- A resource might be a collection of objects or an individual object.
- A resource is identified by URI (Uniform Resource Identifier).
- A resource is manipulated through CRUD (Create, Read, Update, Delete) – operations, which are usually mapped to HTTP methods/verbs POST, GET, PUT, DELETE
- A REST API endpoint is defined by a combination of a resource URI and an HTTP verb that manipulates it.

Last bullet point means that an API is a set of resources with a set of actions you can take on them!

<u>Visual representation of what happens on the client side (request) and server side (response):</u>



The above is pure HTTP, nothing to do with REST specifically. REST builds on top of this and relies on this as its uniform interface.

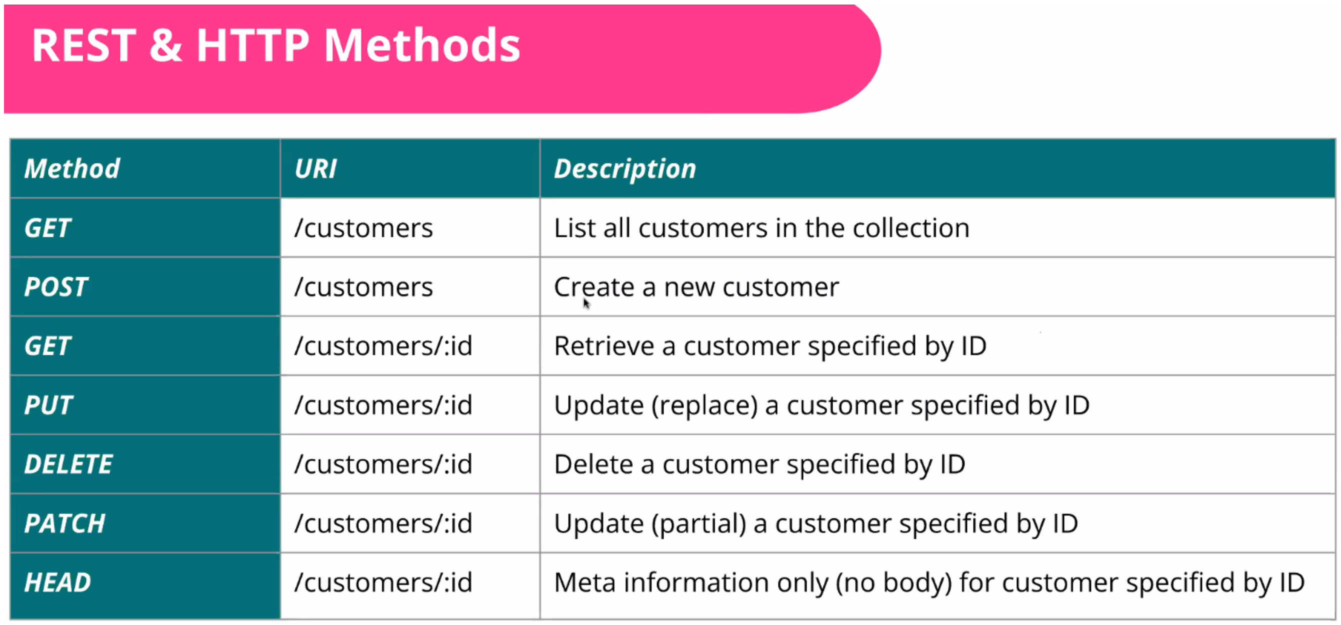


For now, let's just say that if your API is re-defining the HTTP verbs or if it is assigning new meanings to HTTP status codes or making up its own status codes, it is not RESTful.

~ George Reese, The REST API Design Handbook

So, use HTTP verbs as they were initially intended! Don't write a back-end that lets a GET request make an update to the server.

REST & HTTP Methods:



REST & HTTP Methods

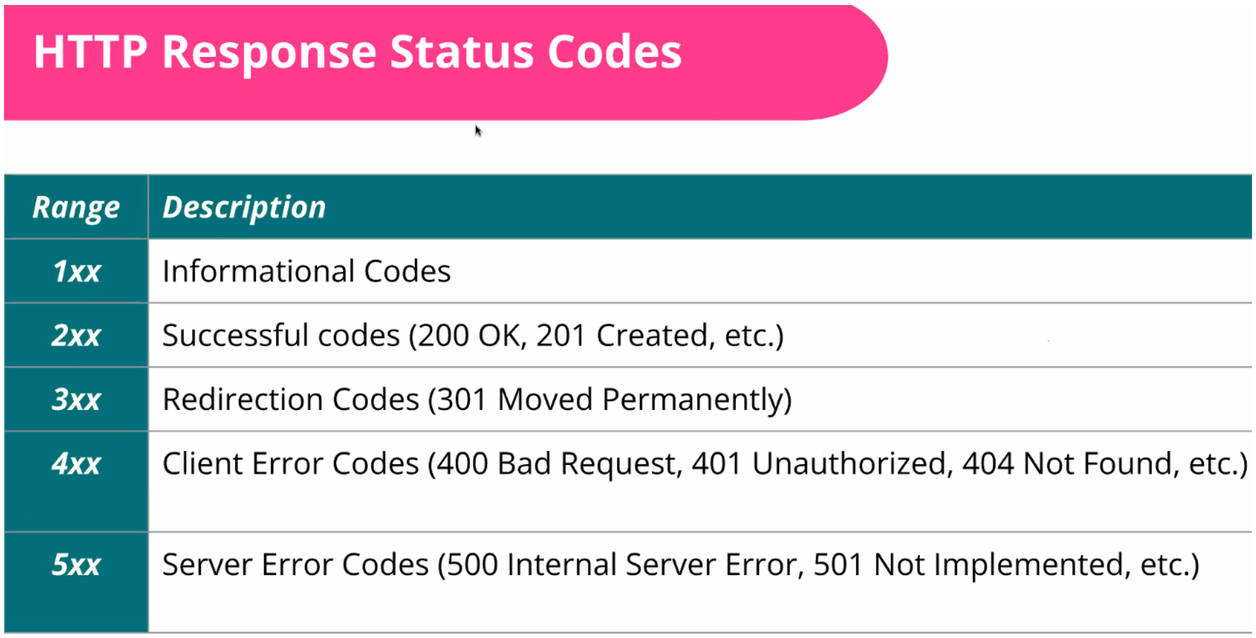
Method	URI	Description
GET	/customers	List all customers in the collection
POST	/customers	Create a new customer
GET	/customers/:id	Retrieve a customer specified by ID
PUT	/customers/:id	Update (replace) a customer specified by ID
DELETE	/customers/:id	Delete a customer specified by ID
PATCH	/customers/:id	Update (partial) a customer specified by ID
HEAD	/customers/:id	Meta information only (no body) for customer specified by ID

The above are the conventions for each method.

Remember PATCH is only to change a bit of the data, PUT is for replacing the entire customer

HEAD – like the lightweight version a GET request – just gets meta info instead of fetching whole thing. Not used very much! Nadeem adds that HEAD requests can be used for things like “pre-flight check” on your API – i.e. checking the GET request works by just getting a status code back (200 = success for example).

<u>HTTP Response Status Codes:</u>



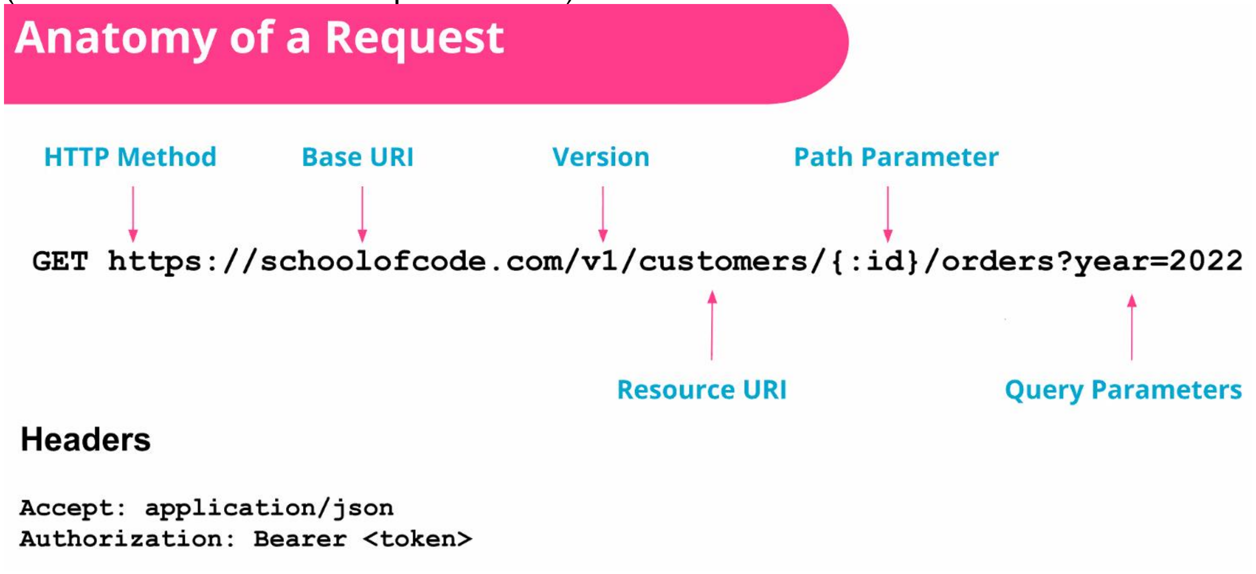
HTTP Response Status Codes

Range	Description
1xx	Informational Codes
2xx	Successful codes (200 OK, 201 Created, etc.)
3xx	Redirection Codes (301 Moved Permanently)
4xx	Client Error Codes (400 Bad Request, 401 Unauthorized, 404 Not Found, etc.)
5xx	Server Error Codes (500 Internal Server Error, 501 Not Implemented, etc.)

You need to consider what response codes you are going to send back for different outcomes to each request, so you don’t confuse your client.

<u>Deeper dive in to the anatomy of a request:</u>

(HTTP method arrow should point to GET)



```
GET https://schoolofcode.com/v1/customers/{:id}/orders?year=2022
```

Headers

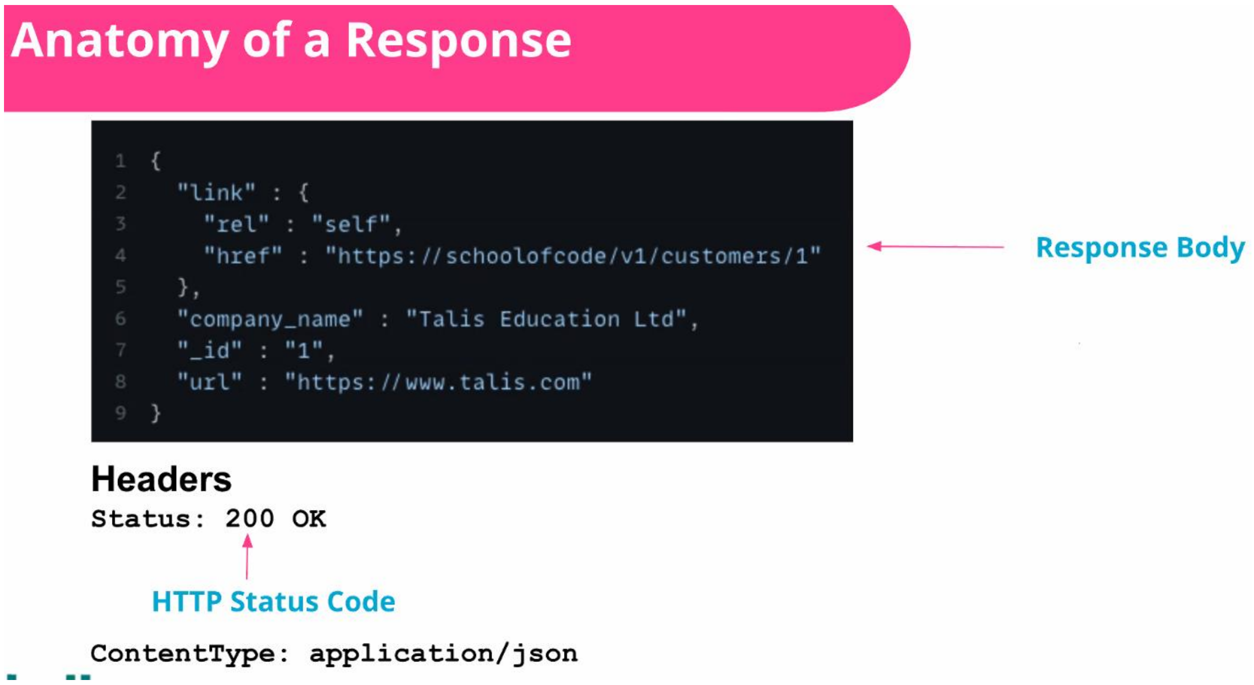
```
Accept: application/json
Authorization: Bearer <token>
```

You should increment versions when making major changes so you don't break clients' apps if they are built around your API. See versioning slide below.

This particular query parameter is filtering orders by year for a specific customer id provided in the path parameter.

In the header - the "accept" is telling the server what type of data you want back. Authorization contains any authentication requirements.

Anatomy of a response:



Anatomy of a Response

```
{
  "link" : {
    "rel" : "self",
    "href" : "https://schoolofcode/v1/customers/1"
  },
  "company_name" : "Talis Education Ltd",
  "_id" : "1",
  "url" : "https://www.talis.com"
}
```

Headers

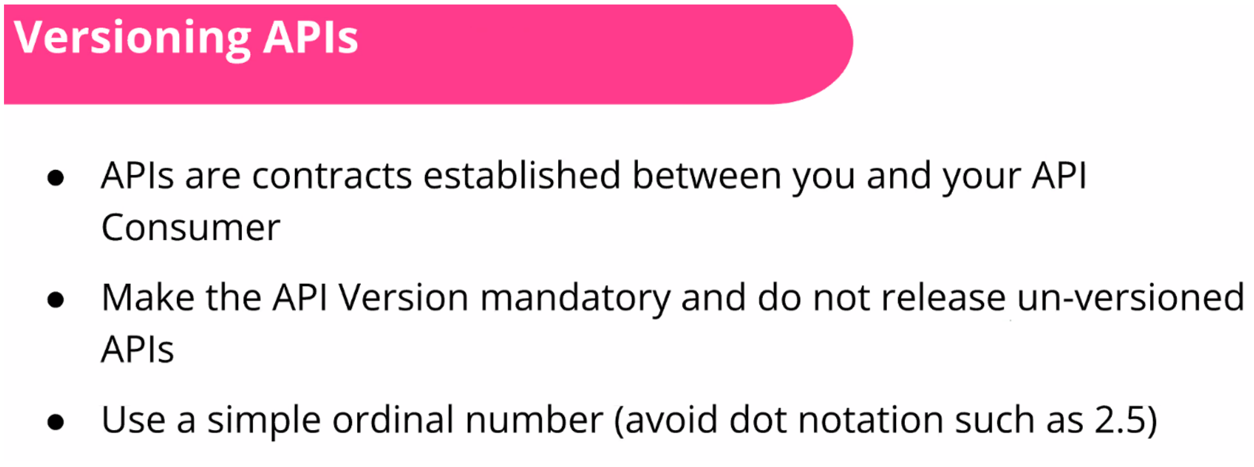
```
Status: 200 OK

ContentType: application/json
```

Response Body
HTTP Status Code

ContentType in Header is important so the server is telling you what type of data is in the body. This is because although you may have specified an "accept" format in your request, the server may not process it. So you can also use this to inspect you got back what you expected (when testing!).

In things like Express Server – a lot of the heavy lifting is handled for you, but you need to understand the above.

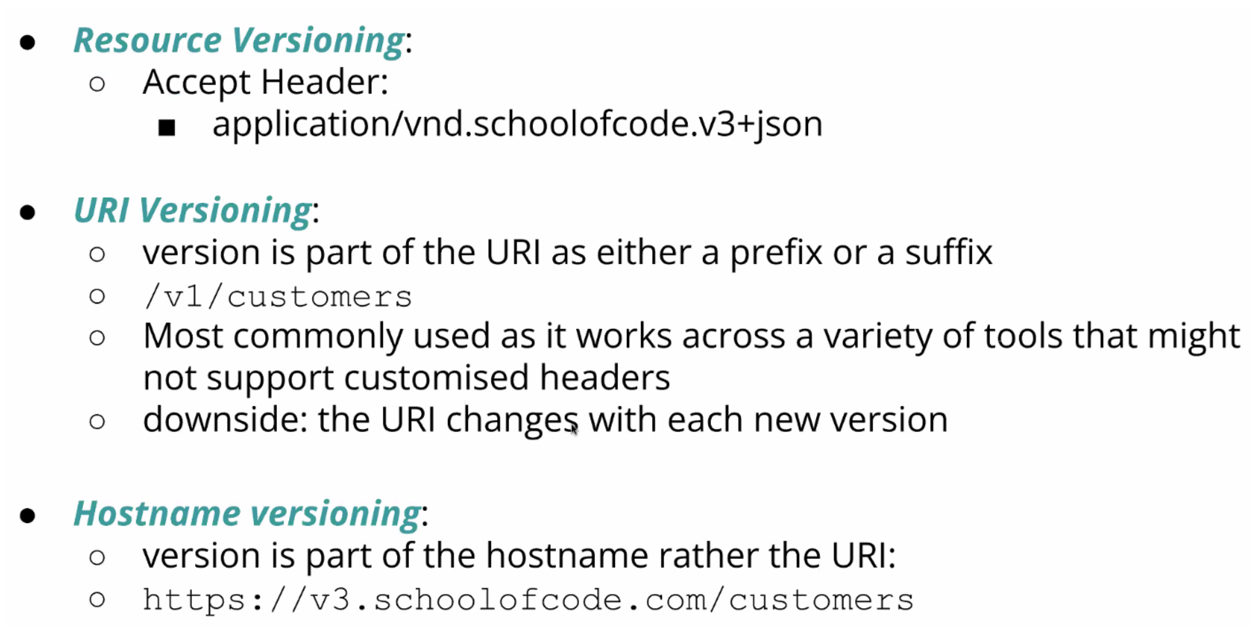
Versioning APIs:



Versioning APIs

- APIs are contracts established between you and your API Consumer
- Make the API Version mandatory and do not release un-versioned APIs
- Use a simple ordinal number (avoid dot notation such as 2.5)

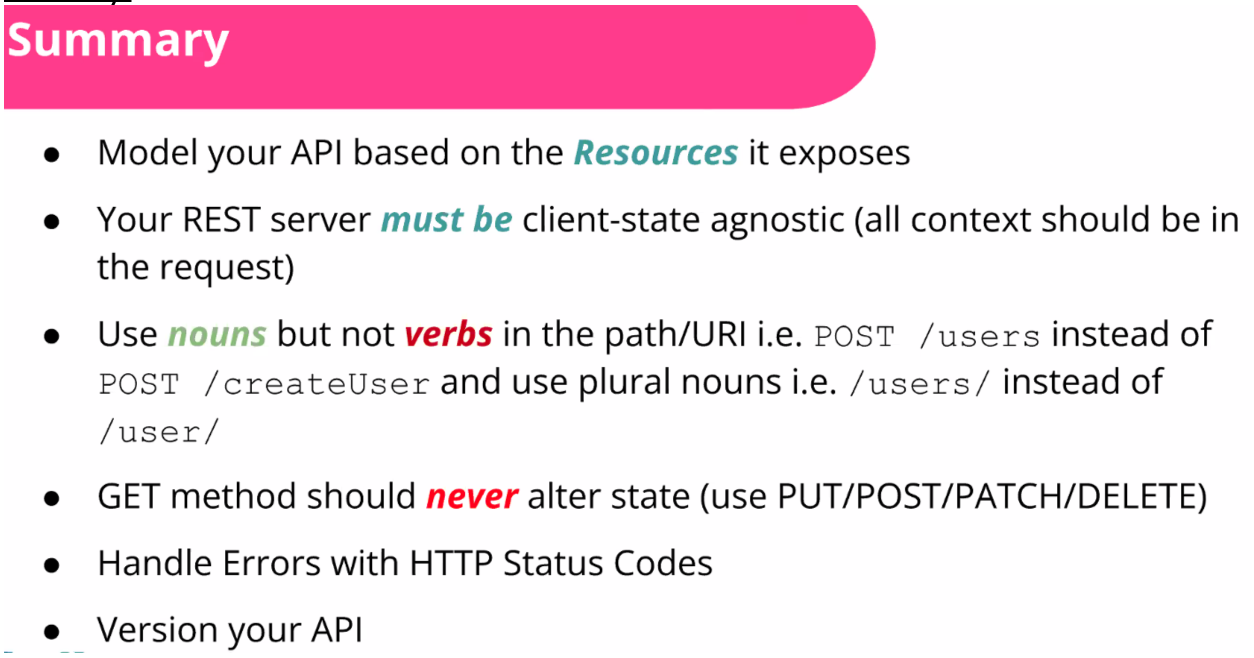
You can implement this in various ways, **but they suggest putting it in the URI**:



- ***Resource Versioning***:
  - Accept Header:
    - application/vnd.schoolofcode.v3+json
- ***URI Versioning***:
  - version is part of the URI as either a prefix or a suffix
  - `/v1/customers`
  - Most commonly used as it works across a variety of tools that might not support customised headers
  - downside: the URI changes with each new version
- ***Hostname versioning***:
  - version is part of the hostname rather the URI:
  - `https://v3.schoolofcode.com/customers`

Versioning is powerful (particularly in URI/Hostname) because V1 could be Node.JS API using Express, V2 could by Python, so client never cares what system is working on the server side and you are not tied to specific technologies.

Summary:



Summary

- Model your API based on the ***Resources*** it exposes
- Your REST server ***must be*** client-state agnostic (all context should be in the request)
- Use ***nouns*** but not ***verbs*** in the path/URI i.e. `POST /users` instead of `POST /createUser` and use plural nouns i.e. `/users/` instead of `/user/`
- GET method should ***never*** alter state (use PUT/POST/PATCH/DELETE)
- Handle Errors with HTTP Status Codes
- Version your API

Selected Q & A and tips

Consider the experience you want the user to have and whether you need to be writing your own socket connections etc. (when talking about UDP/TCP vs RESTful)

Advice for new developers when building RESTful APIs? = The actual presentation above! Don't try and do too much in one request (keep them separate as per HTTP methods above). Nadeem adds – keep it as simple as you can but also, you're working with something that has become its own standards. The hard work in making it a convention etc has already been done for you – so use the libraries that are out there to help you do the heavy lifting.

Is there standard implementation for caching and rate limiting? = It's a balancing game – dependent on your infrastructure and how many requests you are getting. One of the biggest problems you often see is premature optimisation – don't try and build something perfect that will scale indefinitely. If you haven't got a million users, you don't need to build an API that can serve a million requests. Start simple then you can worry about optimising/scaling – as Nadeem can guarantee the 5th time you do it won't be perfect either! This is as true for APIs as any other code!

"Solve the problem that's in front of you"

"Don't try and boil the ocean"

"Great developers are lazy developers"

“Good code reviews (from pull requests) aren’t about syntax etc. Pull requests should be about what the problem is and how it is being addressed. If there isn’t something fundamentally wrong with the approach, then it’s ok.”