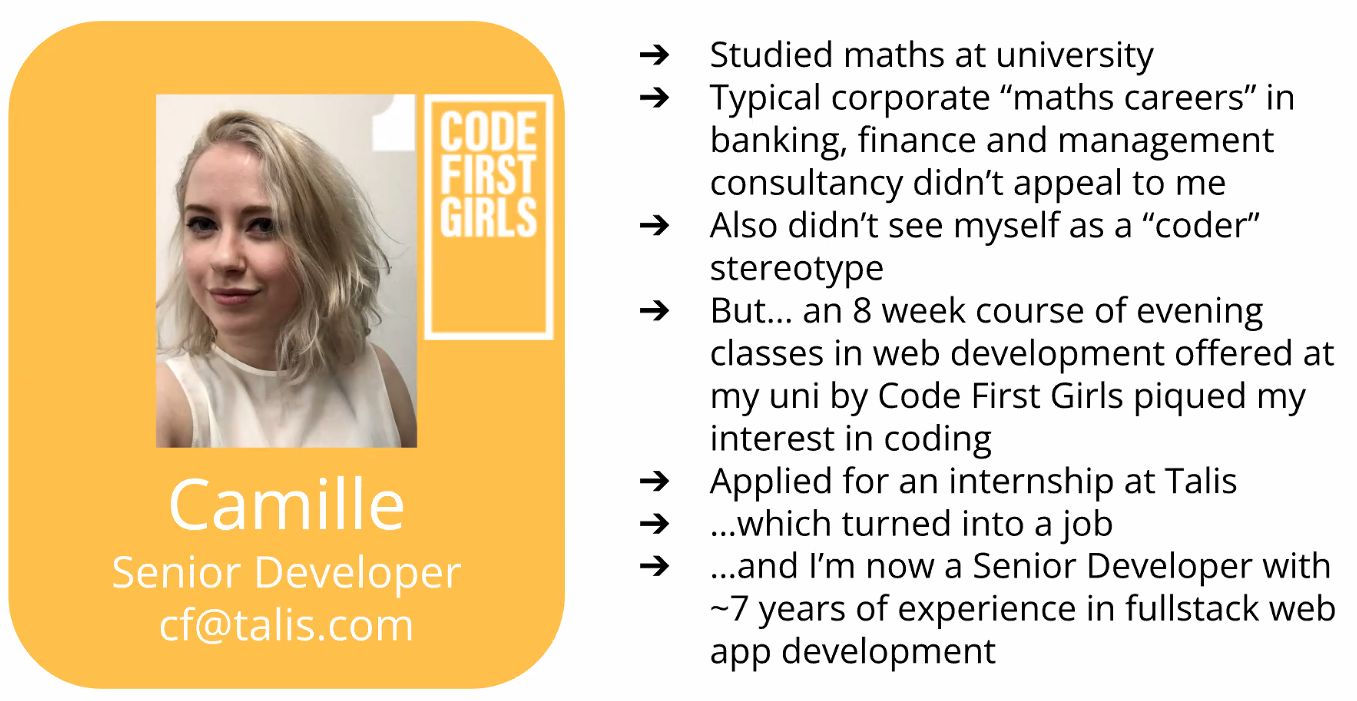
# Nadeem Shabir & Camille Fenton

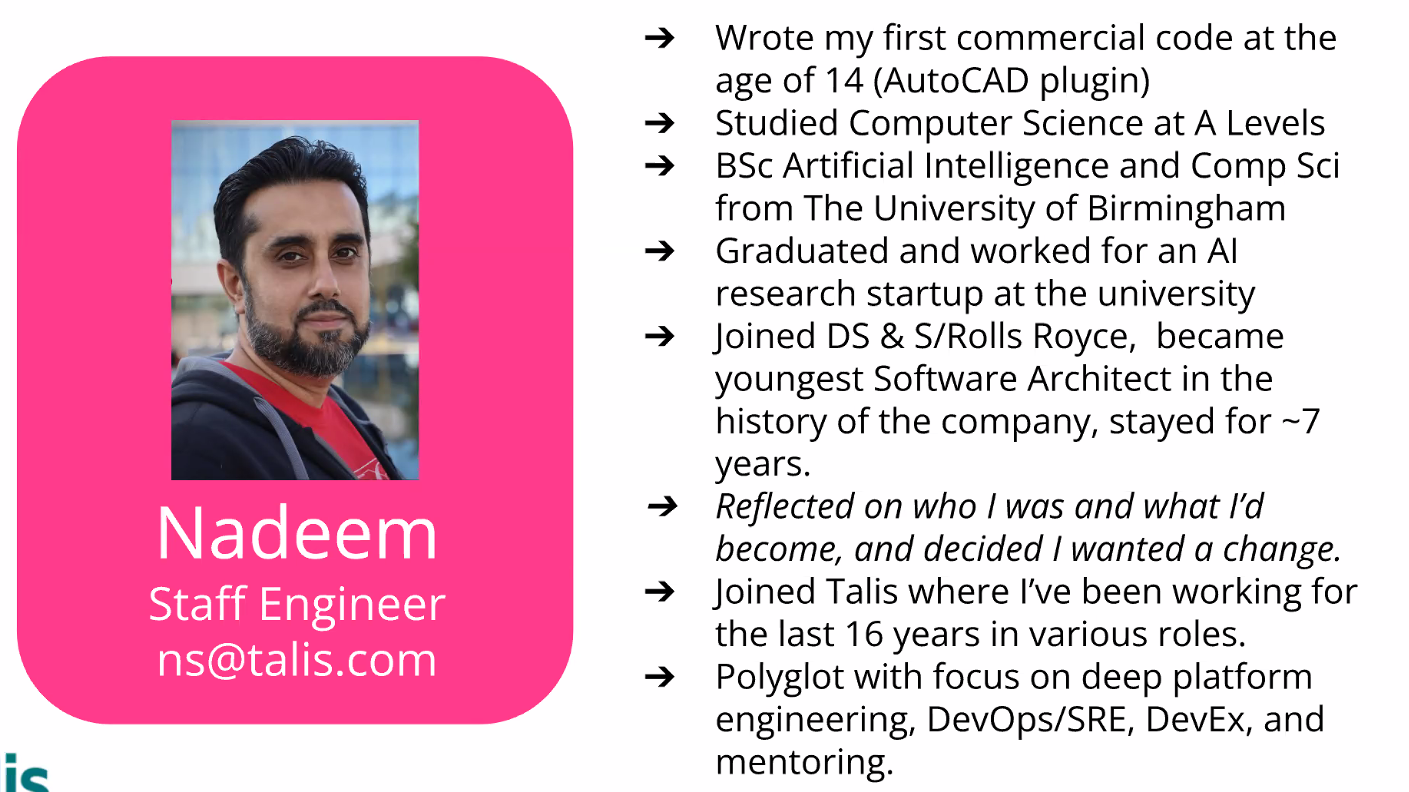
16/6/22

## Introduction to RESTful APIs

### About the speakers:

Both work at Talis.





### 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:

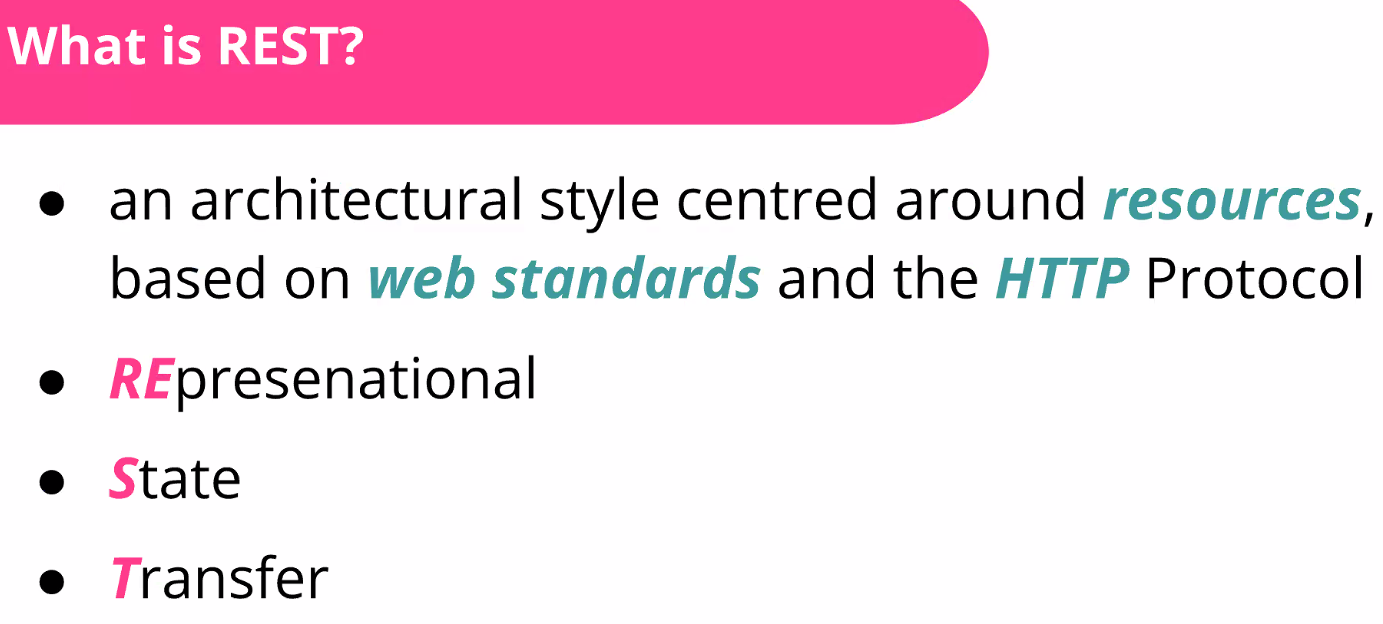


Their philosophy is to try and use the technology that is best to solve the problem in front of you, rather than sticking with a specific language.

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

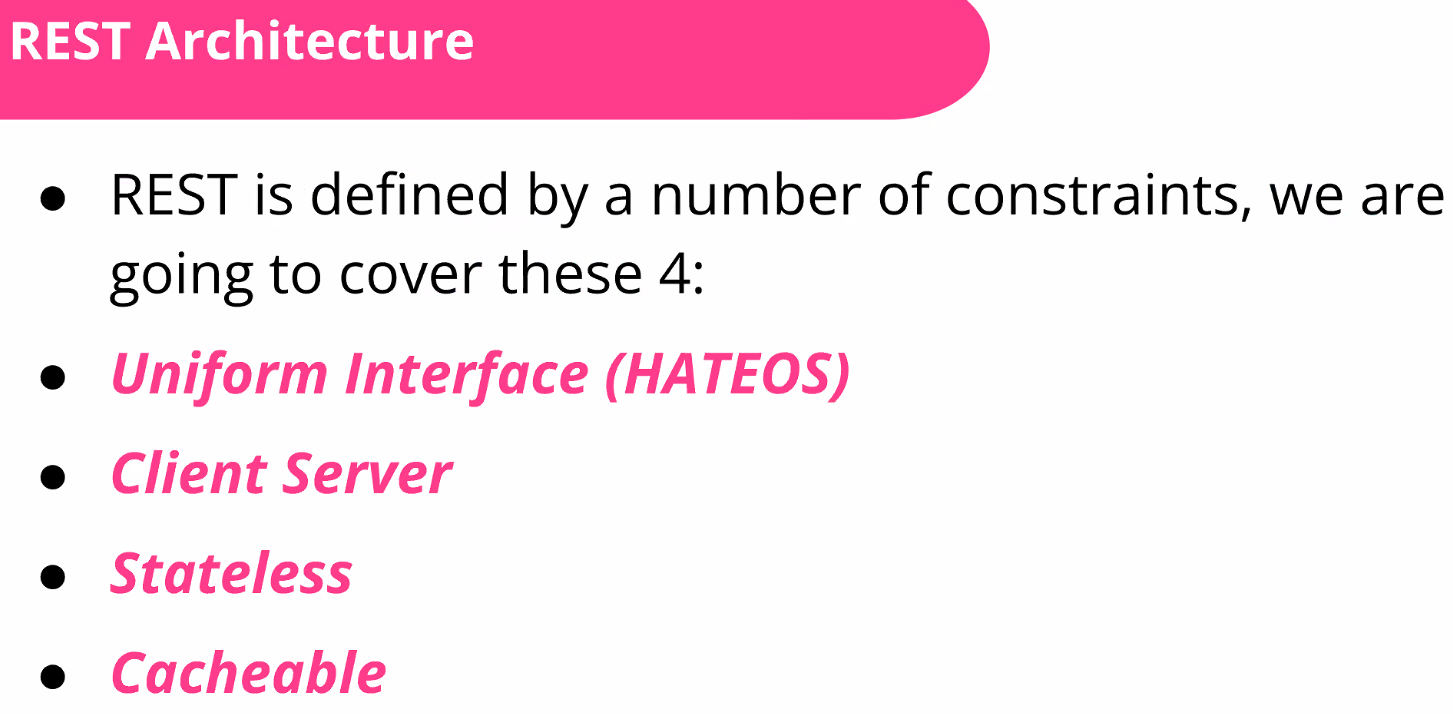
### What is Rest:



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



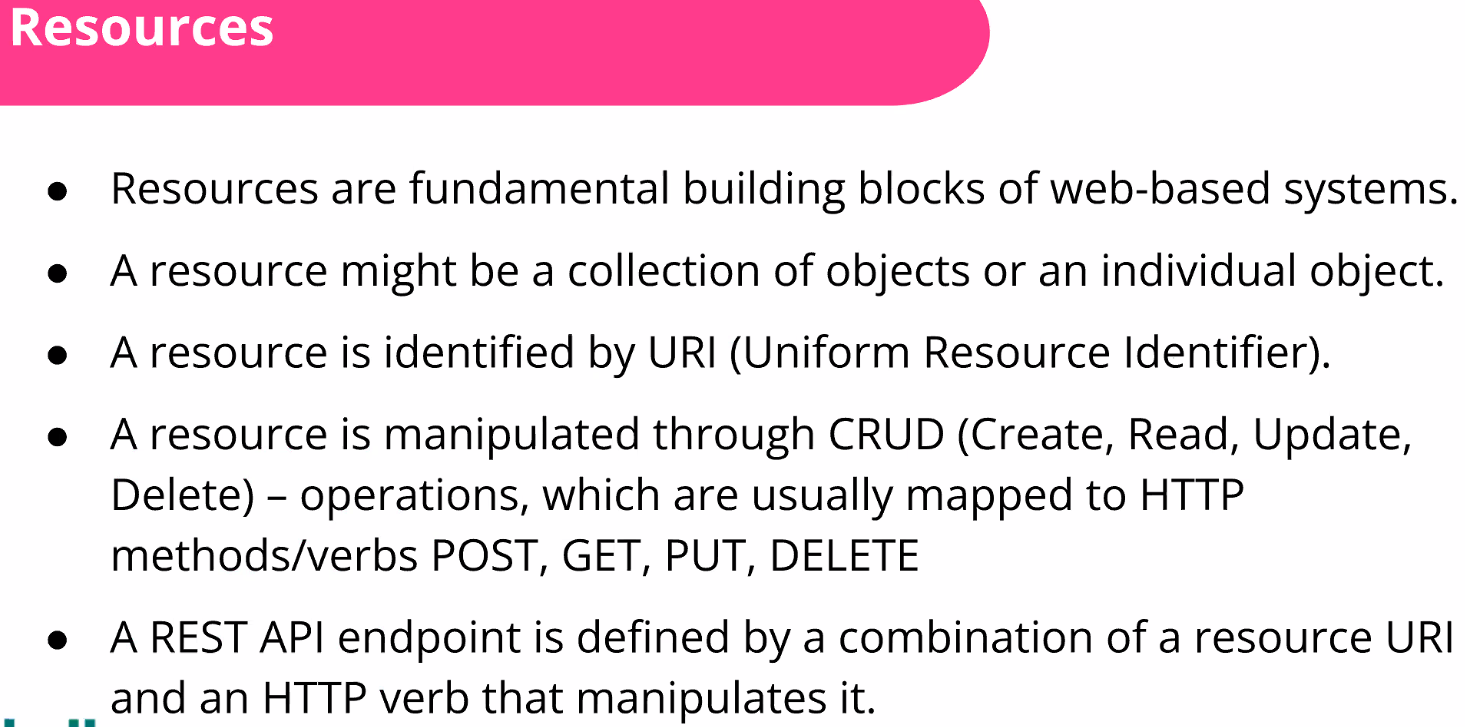
HATEOS – should be [HATEOAS](https://restfulapi.net/hateoas/) - Hypermedia as the engine of application state – means you have a uniform interface for interacting with your resources. So each resource is identified by a URI

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”.

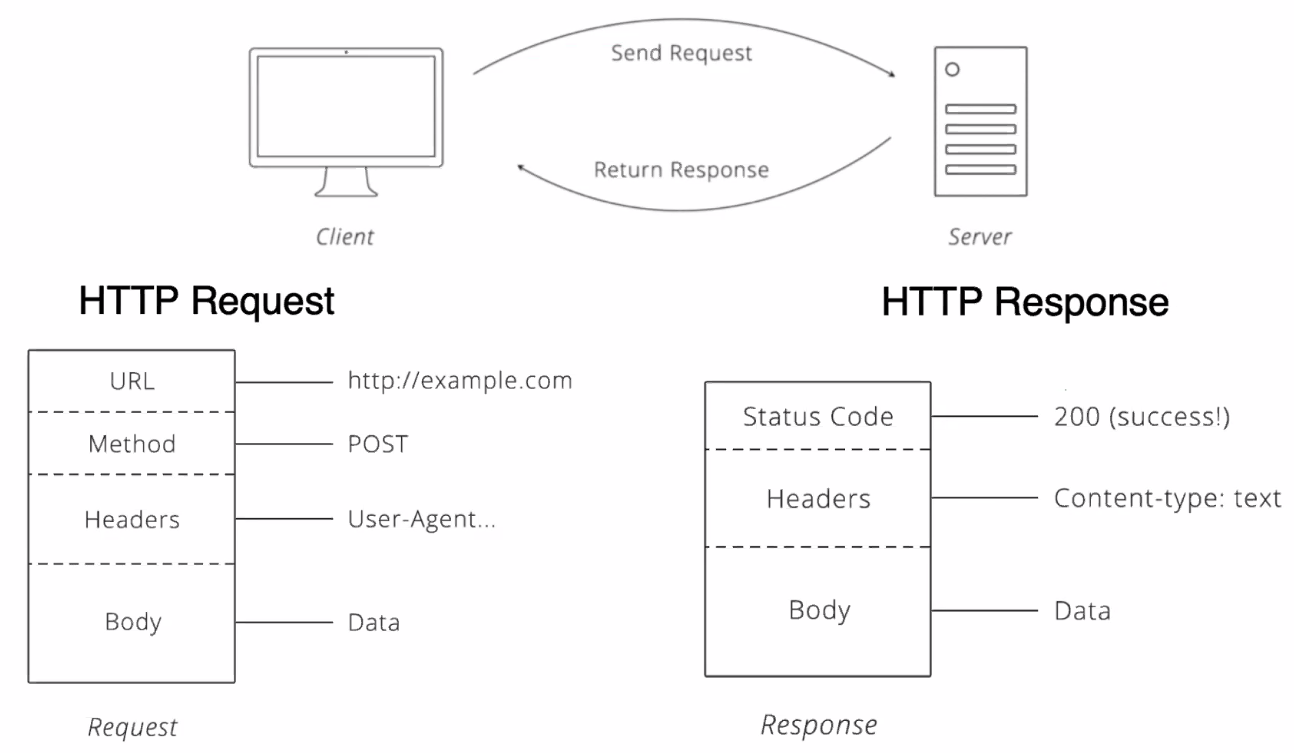
Cacheable – When you make a request, you know every time you are going to get the same response to that request. So doesn’t depend on a previous request. This means you can cache a response and it will always be valid. Caching means server doesn’t have to run a full DB query every time, until a certain “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:

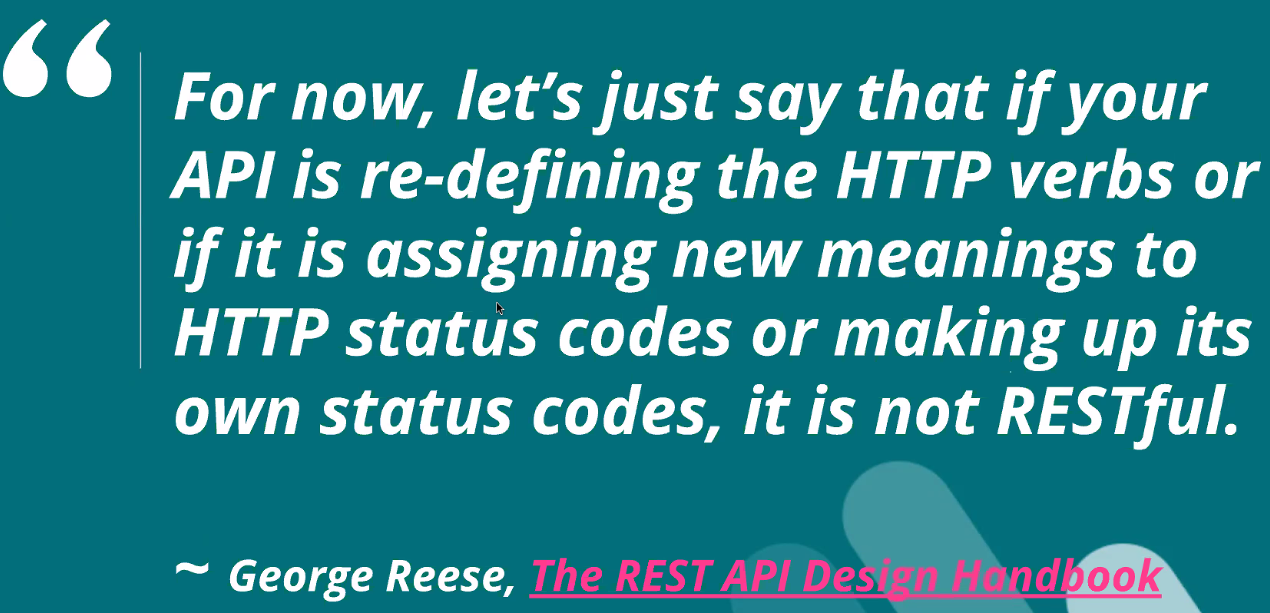


Last bullet point means that an API is a set of resources with a set of actions you can take on them!

### Visual representation of what happens on the client side (request) and server side (response):

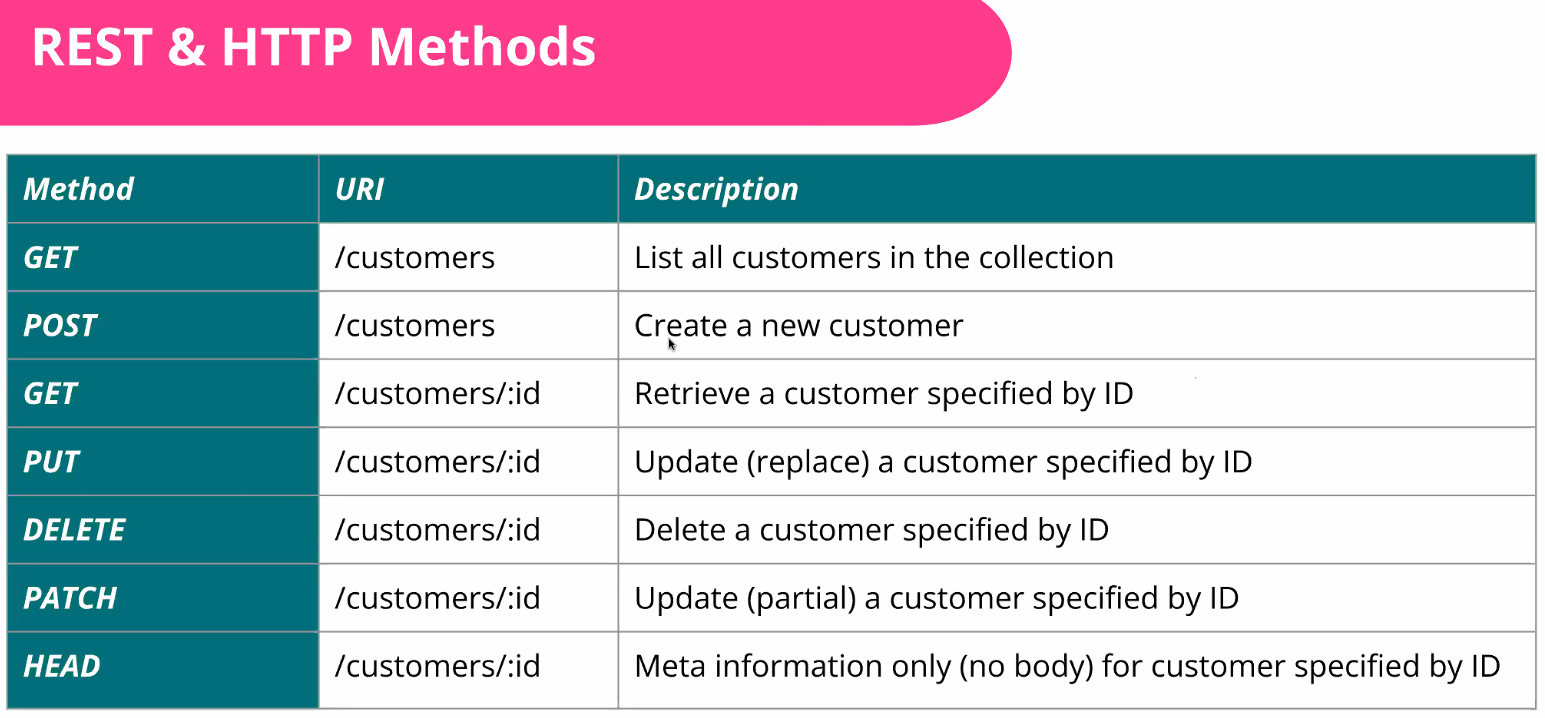


The above is pure HTTP, nothing to do with REST specifically. REST builds on top of this and relies on this as its uniform interface.



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:

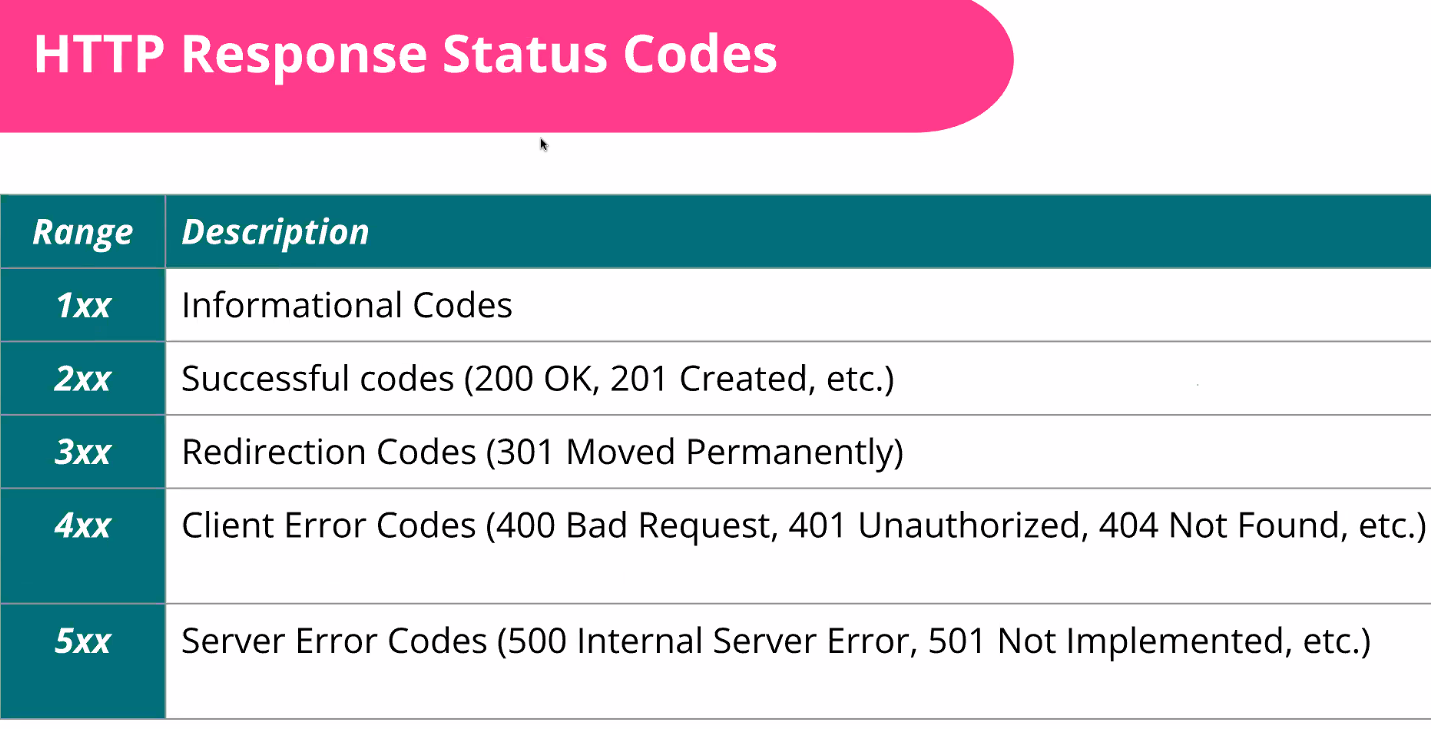


The above are the conventions for each method.

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).

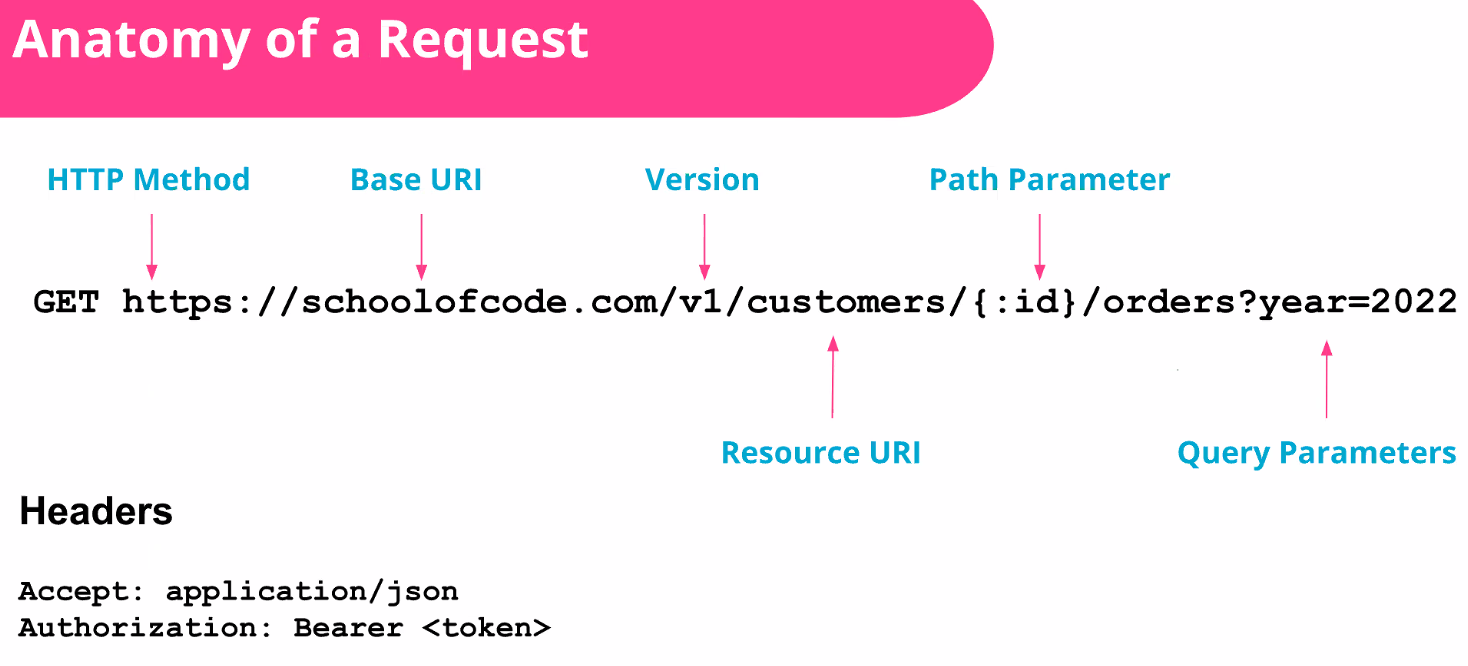
### HTTP Response Status Codes:



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.

### Deeper dive in to the anatomy of a request:

(HTTP method arrow should point to GET)

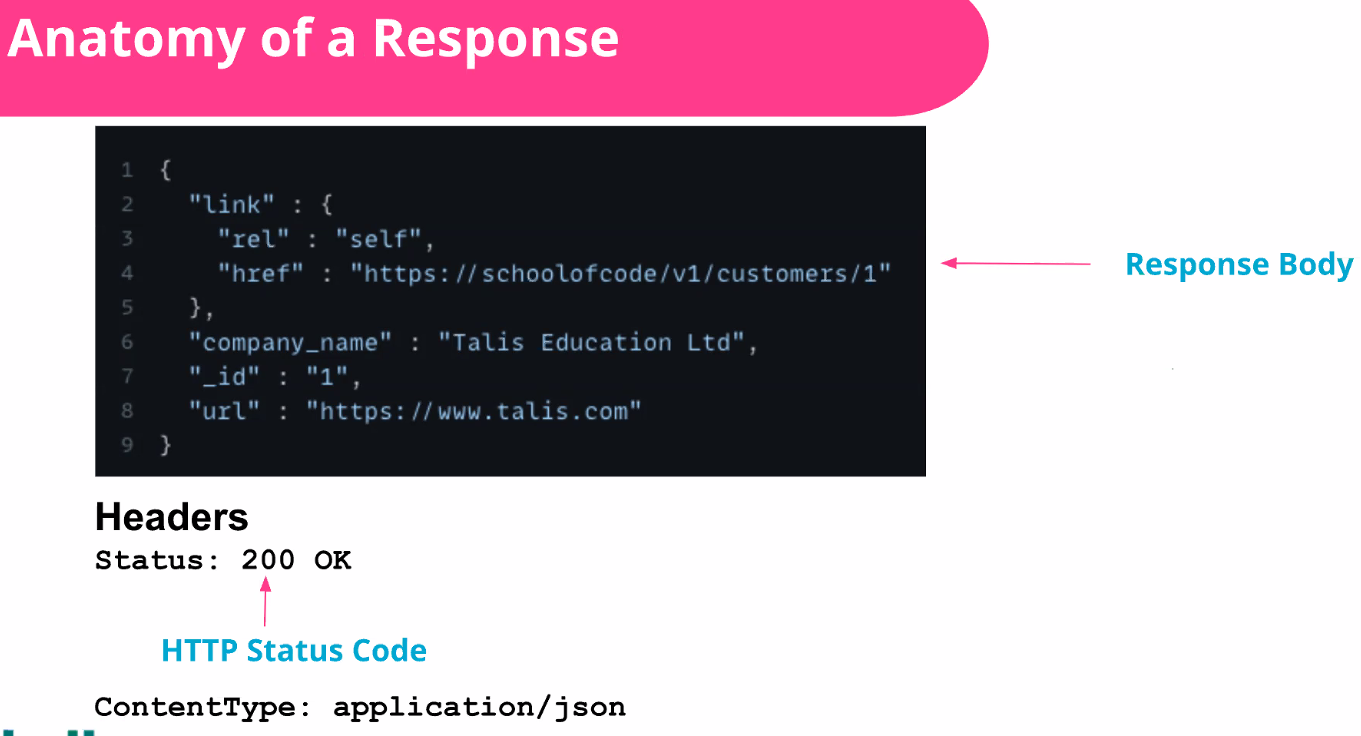


You should increment versions when making major changes so you don’t break clients’ apps if they are built around your API. [See versioning slide below.](#_Versioning_APIs:)

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.

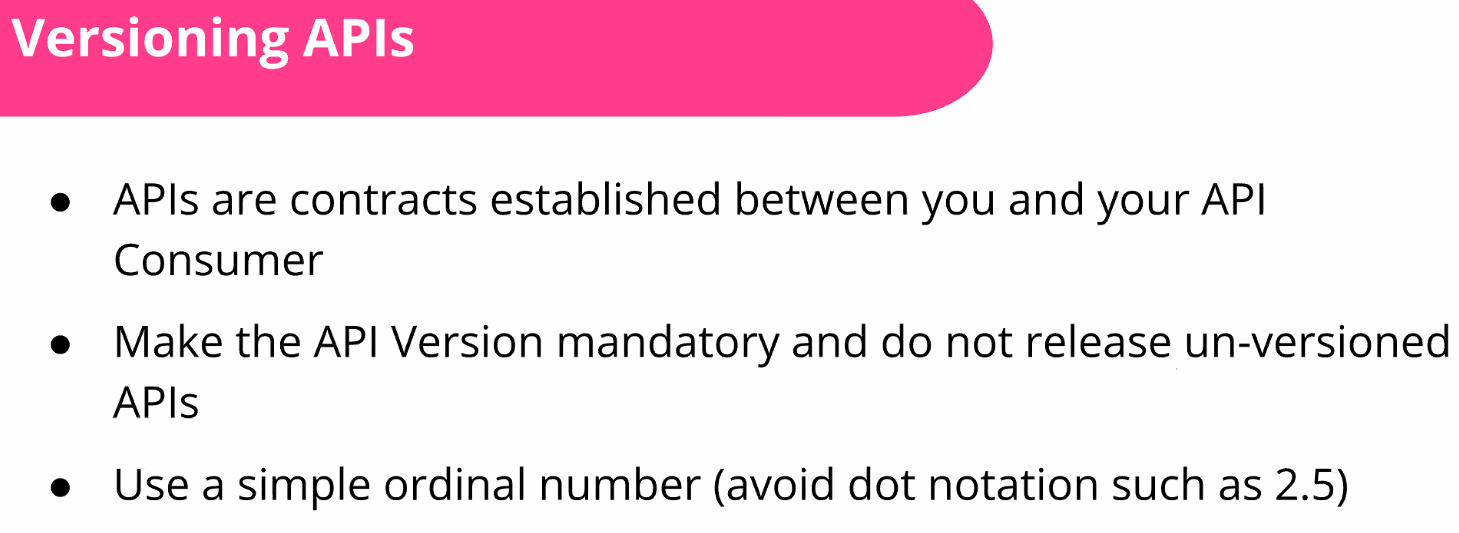
### Anatomy of a response:



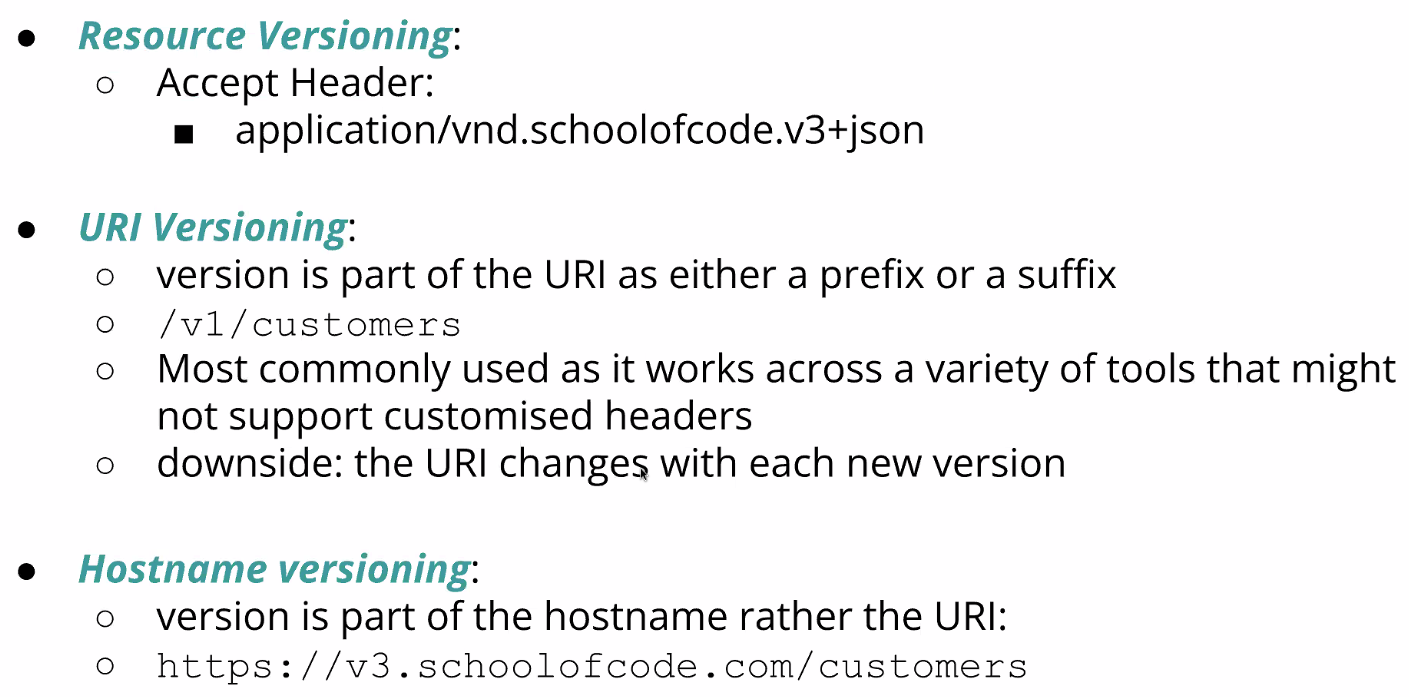
ContentType in Header is important so the server is telling you what type of data is in the body. This is because although you may have specified an “accept” format in your request, the server may not process it. So you can also use this to inspect you got back what you expected (when testing!).

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

### Versioning APIs:

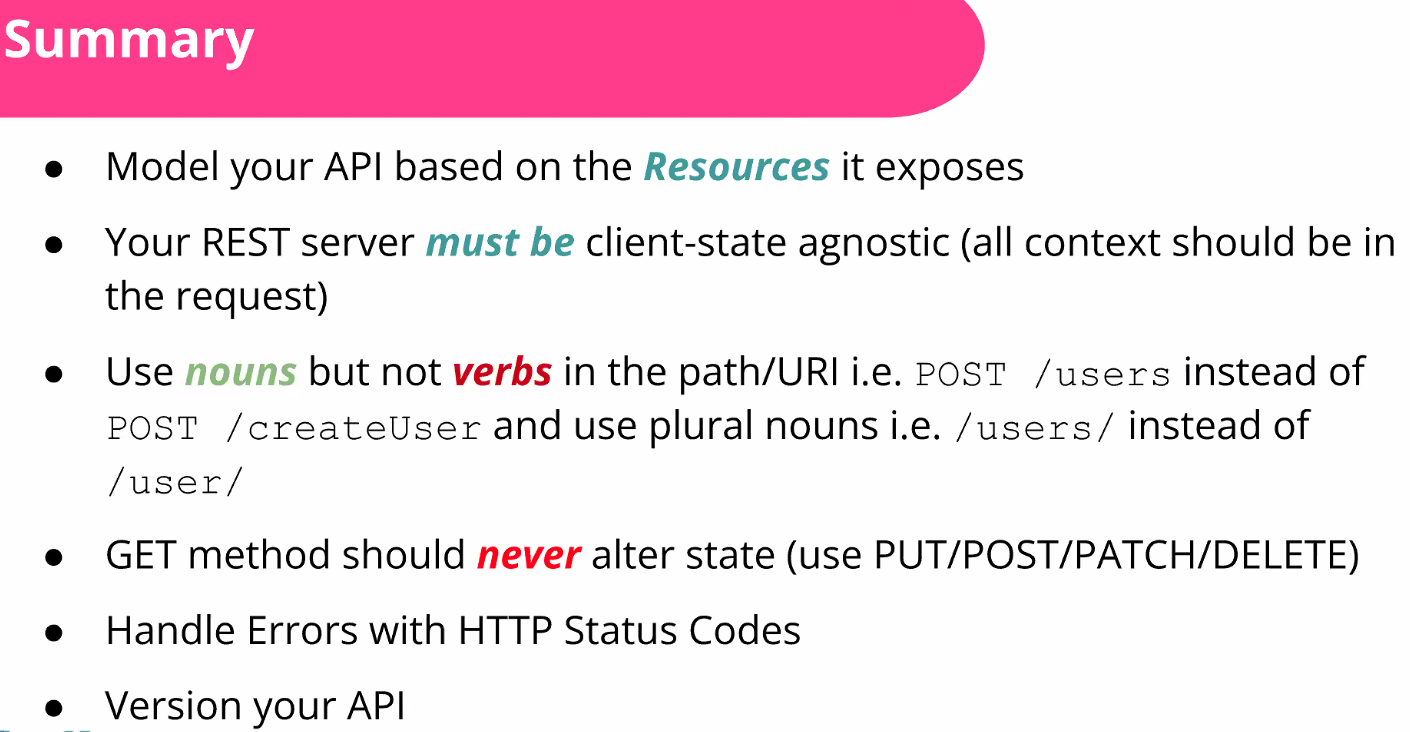


You can implement this in various ways, **but they suggest putting it in the URI**:



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:



## 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](#_REST_&_HTTP) 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.”