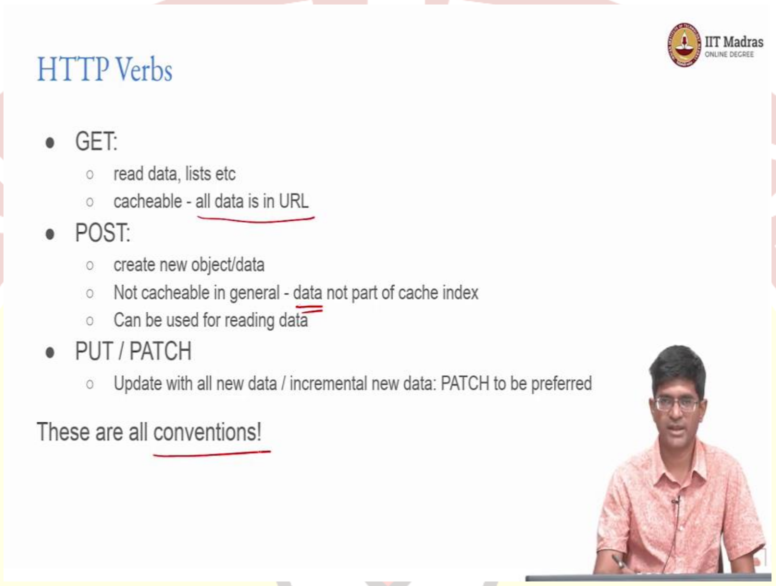
IIT Madras

ONLINE DEGREE

Modern Application Development - II
Professor Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Designing APIs - II

(Refer Time Slide: 00:14)



HTTP Verbs

- GET:
    - read data, lists etc
    - cacheable - all data is in URL
- POST:
    - create new object/data
    - Not cacheable in general - data not part of cache index
    - Can be used for reading data
- PUT / PATCH
    - Update with all new data / incremental new data: PATCH to be preferred

These are all conventions!

Hello, everyone, welcome to Modern Application Development, part 2. Now, coming to HTTP verbs, I already mentioned that you want your URLs as far as possible to be focused around the entities the nouns, which means that you can assign certain meanings to the verbs that are present and are already defined as part of the HTTP standard.

And the usual idea out there is that GET is meant for read data, when you want to read either a list of data or something else, you just specify it as part of the query URL. And the GET basically goes gets all the information and returns it back to you. And because all the data is present in the URL, it largely means that GET requests are cashable.

Because, once again, the convention is that a GET request is not supposed to modify the database, which means that one GET request, and another GET request for the same URL, after some time, should give me back exactly the same data unless something changed on the server, which makes it very easy to cache, because all I need to do is keep track of the URLs handled by GET. And some way by which I can validate or invalidate the cache if needed. So, the cache memory can basically say, if somebody else has gone and modified the database, in the meantime, invalidate these URLs, or invalidate the entire cache.

But otherwise, if I have mostly only been reading from the cache, next time I get a GET request for a certain URL, just take it from the cache, do not bother hitting the database at all, send back the response. Which means that posts for, on the other hand, because fundamentally, a post does not have much of a URL, it just has the endpoint to which you are trying to connect.

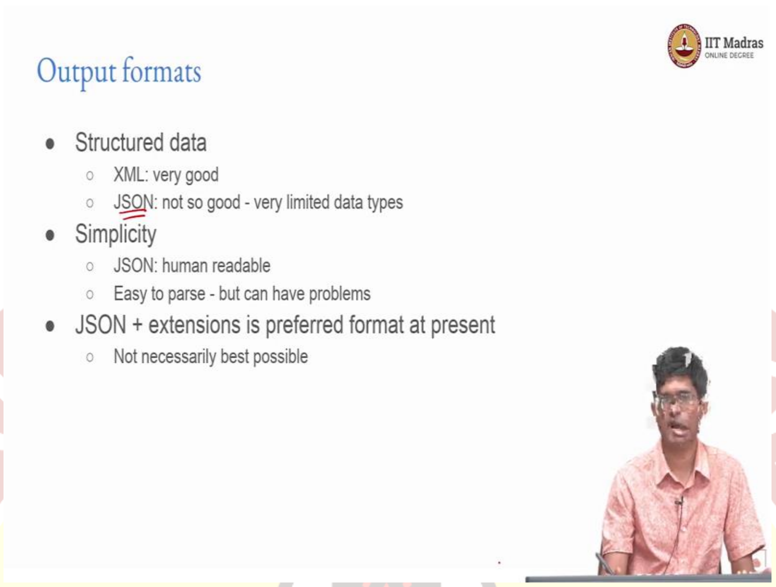
But the rest of the information corresponding to the post is present in the post body, the request body. And therefore, it sort of makes sense to use a post in order to actually create new data, you take whatever is coming in the request body and use it to create a new object in your database and return some kind of reference to that object.

Now, what that means is because you are using the same URLs, the data is not part of the cache index, the data is not getting cached. And therefore, in general, POST requests are not cacheable. Now, keep in mind that a GET request, in principle could go and modify the database. Similarly, a POST request could be used purely for Read Only purposes. But conventions are that, that is not the way you use them. When you are doing read only use GET as far as possible, when you want to create something new use POST.

And then, of course, there are other verbs, put, patch, delete, and so on. Each of them has a specific kind of meaning associated with it. At the risk of repeating myself too many times, I am going to say it once again, these are conventions, keep that in mind when you are actually going through a document of an API. That could be nasty surprises at some point that you do not expect, which probably means that whoever wrote the API has done a bad job.

Because breaking a convention is one of the worst things you can do as an API designer. It means that a person is expecting a certain kind of behavior and you are intentionally telling them that oh, look, I changed it. And that is not the way people are, people are not normally looking for such trip ups. So, they might miss that, assume a certain kind of behavior, do something and GET the wrong response altogether.

(Refer Time Slide: 03:45)



Output formats

- Structured data
  - XML: very good
  - JSON: not so good - very limited data types
- Simplicity
  - JSON: human readable
  - Easy to parse - but can have problems
- JSON + extensions is preferred format at present
  - Not necessarily best possible

Now, what about output formats? what kind of output format should you use when you are constructing an API, the important thing to understand over here is you are usually trying to return some kind of structured data back to the client. And what I mean by structured data is you could of course, just return the entire HTML that they need to see. Ultimately, that is what a flask kind of system does. But when flask is operating an API mode, you are not really rendering the HTML and giving back, you do not want to do that, because you want to give the front end the flexibility to render in whatever way it likes.

Now, XML, as one of the original markup languages is actually very good at this, meaning that it can handle structured data very well. It can, clearly indicate hierarchies. It can associate types with data, and you can have a very well-constructed valid XML document that is very easy to parse. The problem is it is also very verbose, and in a lot of cases, it is overkill. You do not really need that much complexity.

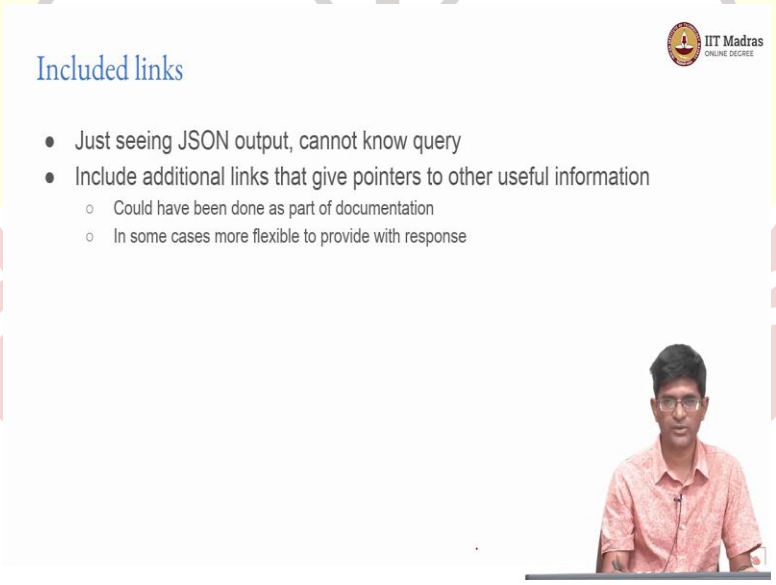
Therefore, the popular one today is JSON, JavaScript Object Notation. Now, it has problems in the sense that, for example, you there has strict limitations on the types of different data that you can specify. And structured data, for example, specifying that something is a type of class cannot be directly indicated in the JSON itself, you need to have intelligence in the system that actually tells you that, yes, this is supposed to be interpreted as a specific type of data.

But it is very simple. It is human readable, you can literally almost write it out by hand, and it is also very easy to parse in most languages. Although there can be occasional trip up. So there, you have to be a little bit careful about how you go about parsing. But overall, the consensus is that JSON plus a few extensions of it are the preferred format, at present. And I am stating that word at present for a specific reason.

It is because one of the other things that you need to keep in mind over here is, all of these are sort of the best practices and conventions that exist today. A few years ago, they were not considered best practices, XML was definitely considered superior because it had the ability to handle structured data, but the simplicity of JSON won out over time.

Is it possible that some other language will become more popular or some other way of representing things will more become more popular later on in time? Entirely possible. So, keep that in mind, but as far as possible in today's environments, it is preferable to use the languages and formats that are currently popular simply because they are easy. And once again, people will be expecting certain kinds of behavior from them.

(Refer Time Slide: 06:37)

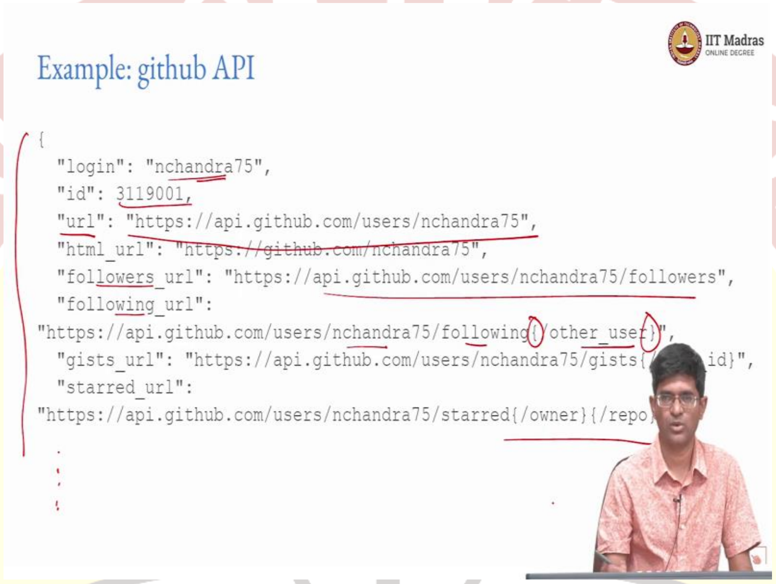


Included links

- Just seeing JSON output, cannot know query
- Include additional links that give pointers to other useful information
  - Could have been done as part of documentation
  - In some cases more flexible to provide with response

Now, one bit about including links, I would mentioned this a little bit earlier. There is a sort of recommendation to include links to various other API endpoints or URLs in the response that you give as part of an API request. One of the reasons is, if I just see the JSON output corresponding to a query, I do not really know what the request was that led to that query. So, having a sort of self-link, might seem redundant. But it is actually useful if you are sort of trying to debug or record this for later. And in addition to that, you could give pointers to other useful information.

(Refer Time Slide: 07:16)



Example: github API

```
{
  "login": "nchandra75",
  "id": 3119001,
  "url": "https://api.github.com/users/nchandra75",
  "html_url": "https://github.com/nchandra75",
  "followers_url": "https://api.github.com/users/nchandra75/followers",
  "following_url":
"https://api.github.com/users/nchandra75/following{/other_user}",
  "gists_url": "https://api.github.com/users/nchandra75/gists{/    id}",
  "starred_url":
"https://api.github.com/users/nchandra75/starred{/owner}{/repo
```

A good example of this is what is followed by the GitHub API, I would mentioned this earlier. If you just go to API dot GitHub dot com slash something. You will GET a response, which is basically a chunk of JSON text, there is more to it. I have left it out over here. The first question is, Oh, I saw this chunk of JSON, I do not know where it came from. But remember what I said about self-links, they do that, they give you the URL that was used in order to GET this information right here.

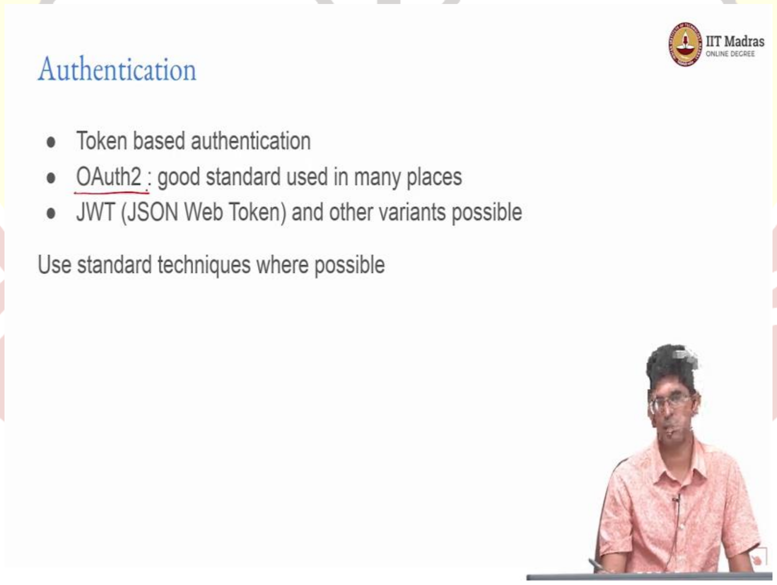
So, if you go back to this URL, you will find exactly the same JSON text again, you also find that there is a unique ID over here, which means that even at some point, if I change my login name, the ID will remain unchanged. So, there should be some way by which I can translate an ID into the login and then GET to the URL that gives me the data that I need.

Or there might be another API request that directly gives me information based on my ID without having to even think about the login name. Now, the interesting thing as you go a bit further down, is that it also gives you things like a follower's URL, so who are all the people following me? on GitHub. And similarly, it will also give me so called following URL, which says that is the user nchandra75, my name, following another user.

So, this would essentially return, it is supposed to return kind of Boolean. The interesting thing is they also make use of the fact that there are specific HTTP status codes that represent success and failure, and return one of those codes to indicate a success or a failure. Which means that I could just construct URLs now look at this notation here with these curly braces, effectively, it is telling me that that is a new parameter to be filled in, it is not part of the URL. But if you fill that parameter with something else, then it would give you further information about whether that user is following nchandra75 on GitHub.

So, in other words, you know, you could go further, you could have multiple things where there are potentially nested such optional parameters and so on. All these URLs, in other words means that if I just go to api.github.com, from there, I can start crawling and seeing, these are the subsequent requests that I get. And I can collect a whole lot of useful information just by following those links. It is sort of a way of discovering what is present in the API.

(Refer Time Slide: 09:43)



Authentication

- Token based authentication
- OAuth2 : good standard used in many places
- JWT (JSON Web Token) and other variants possible

Use standard techniques where possible

Now, what about authentication, In the context of API's, I have not really spoken anything about authentication. It largely follows similar practices to what you would use in the context of securing functions in a flask application, for example, although of course, you might, your back end might be written in other ways, not necessarily only flask. The thing to keep in mind is, how are you going to verify whether someone is logged in or not? You cannot really expect that it is interactive.

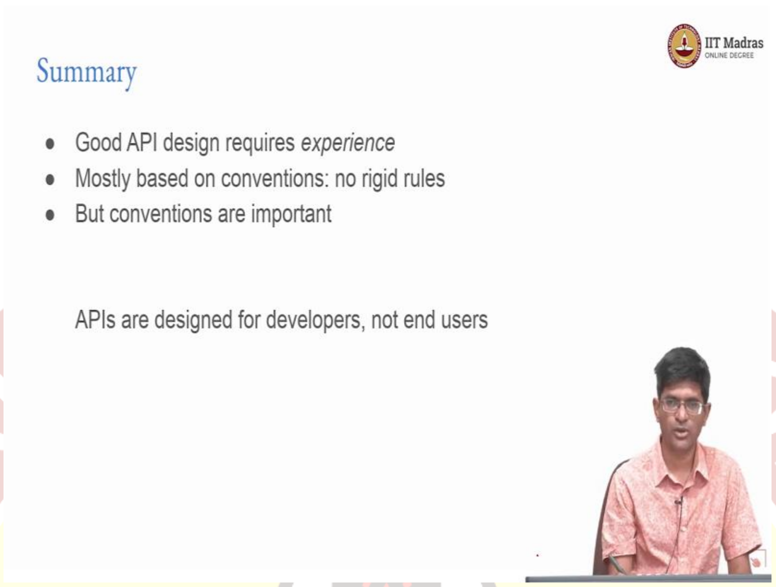
Because at the end of the day, it might, you might, of course, look for cookies, because at the end of the day, you are saying that the API is going to be used by some front end, the front end can therefore take care of setting the cookie, you now need to be careful was the front end that was, where it was being served from? And where the API is running? Are they from the same server does the cookie support or sort of cross origin cookie behavior, or if they are running on the same server, of course, it does not matter.

But there are certain issues that you need to take care of when you are performing authentication, especially with an API back end. Today Oauth2 is probably one of the best standards that can be used. It is seen in many different places for authenticating API's, Ultimately, what happens is that some kind of a token is being set as part of the authentication process, and is then being passed back and forth between the end user that is the front end, and the back end, which is verifying whether the token is present with each request. So, it validates each request before it actually performs the function and gives the response.

There are other techniques, JSON Web Tokens are things that are quite popular these days. This is not something that we can GET into in detail. At the moment, at least. The main point of advice over here is once again, use standard techniques where possible, especially in techniques like authentication, why? Because any mistake that you make over there is a security is a big security problem for your system, it means that you intended to keep something safe and behind an authentication vault, and you actually have a problem with how it was implemented.

Things like Oauth2 and JWT, when properly implemented, have at least been studied very well, by a lot of people. And therefore, there are good reasons to believe that they are secure, at least to the reasonable extent that we know. But if you try to roll your own techniques, there is a good chance that there might be a bug there that you do not catch, and somebody else does after you have deployed it.

(Refer Time Slide: 12:24)



Summary

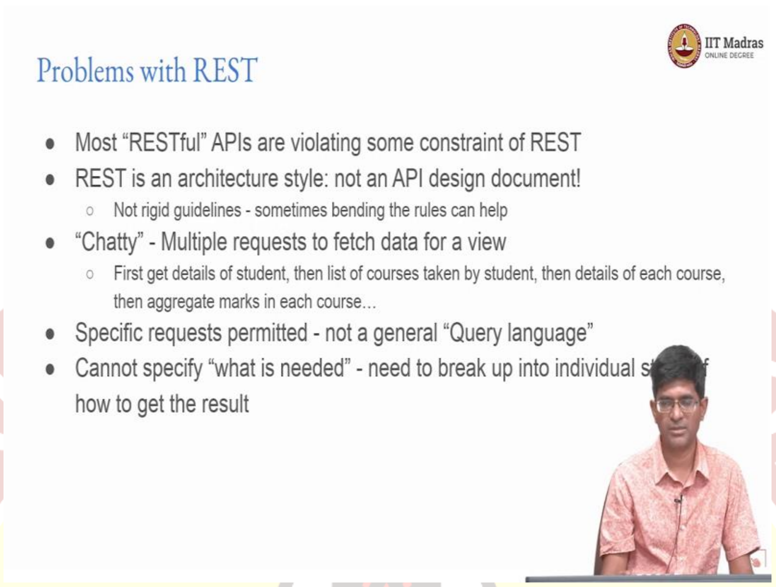
- Good API design requires *experience*
- Mostly based on conventions: no rigid rules
- But conventions are important

APIs are designed for developers, not end users

So, to summarize, the thing is good API design requires experience, there is a lot of work that is required. In order to build this up, you do not just go read a book and then say, I know how to build a good API, you have to try it out, you have to try it with a team of users who are going to be using your API. And over time you build up your own conventions for what really works the best.

Mostly, it is based on conventions, there are very few, if any, rigid rules in the system, But the conventions are important. It is not, once again, something that is just option. If you do not follow the conventions, there is a good chance you have made your system impossible to use. And the key point to keep in mind when designing an API is you are designing an API for a developer to use, not the end user at the front end.

(Refer Time Slide: 13:17)



Problems with REST

- Most "RESTful" APIs are violating some constraint of REST
- REST is an architecture style: not an API design document!
    - Not rigid guidelines - sometimes bending the rules can help
- "Chatty" - Multiple requests to fetch data for a view
    - First get details of student, then list of courses taken by student, then details of each course, then aggregate marks in each course…
- Specific requests permitted - not a general "Query language"
- Cannot specify "what is needed" - need to break up into individual s f how to get the result

So now with all of this in place, one word about certain problems that are there with the REST approach. And one of the first problems is most so-called RESTful API's are violating some constraint or the other of the original REST principles. It is not always obvious what, but there are certain things where people just say that oh, if you are following certain criteria, then this is a RESTful architecture. It maybe it may not be. So, first of all, is RESTful, the same as the rest is REST architecture or is RESTful an approximation of REST architecture is itself unclear. And the problem is, there is a lot of conflicting documentation available online, which can tell you that something either works correctly, or it does not.

Now, in the same context, you need to remember that REST is an architectural style. It is not a design document, it is not something that says, follow these steps and you GET an API. It is not a set of rigid guidelines and in some cases, bending the rules can actually make the API design a lot more friendly and easy to use, then rigidly adhering to the principles of REST. REST was sort of put down as an architectural style for a reason. It was based on a good understanding of what constitutes the web itself.

Having said that, REST could potentially be implemented even on protocols that are not just HTTP. There could be other kinds of protocols, provided they follow similar kinds of behavior, statelessness, catchability, and so on, and write the resource identifiers and so on if they use similar kinds of structures, you could have a REST architecture implement with the different protocols.

So, those are important things to understand. And what you need to understand is you are interested in designing a web API, not necessarily a REST API, if it turns out to be REST, good, because that means that is following fairly good principles or a good architectural style. But if you need to bend it or break it, for some reason, you need to know why you are doing so. And that it is okay to do so.

One of the problems with REST, is that it is sometimes called a chatty protocol, meaning that a lot of back and forth requests are required in order to GET the data corresponding to a single view, sometimes. Let us say that you want to get like the top students in a course first you get the details of a student, then you will get the list of or rather, you want to ge something about their grade point average, you get the details of a student, you take the list of courses taken by the student can then get the details of each course aggregate marks in each course, it is potentially possible that if you do not have a single function on the API that gives you all of this, you will have to do many such requests back to the server in order to collect all the information that you need.

The REST structure also specifies a certain set of requests that are permitted. It is not a general query language. And even something as simple as saying, you know, how do I specify what is needed? let us say that I want to retrieve something about a student, but I am not interested in their date of birth, or their address or their hostel room number, I only want the roll number and their name.

Usually what happens with the kind of REST architecture is that REST API’s is that you put a request for a user, you GET back all the information corresponding to the user, and then you choose what to do with it. But is there a simple way to sort of specify that only wanted their roll number and name? It is not naturally part of the REST architecture.

But, there are a number of API's that you will actually see, including things from Google, where you can specify the fields that you want. Are those following RESTful architectures? Possibly, I mean, why not? I mean, it is not really fundamentally breaking anything in risk to ask for only a subset of the fields, but it means that you are complicating your query structures a lot more.

As you start to look at these kinds of structures in more detail, you will find that there are a number of such special cases that you need to deal with. And that is where it sometimes makes sense to say is there a better way of handling this entire way of making queries from a server? And that is what we are going to look at next.