**Why first-party cookies and third-party cookies are treated so differently?**

*To be fair, third-party cookies aren't any less cookies than first-party cookies. ... The difference between them boils down to what domain created the cookies in the first place. A first-party cookie refers to a cookie created by the domain that a web user is visiting.Jun 8, 2017*

*First-party cookies are stored by the domain (website) you are visiting directly. They allow website owners to collect analytics data, remember language settings, and perform other useful functions that help provide a good user experience.*

*Third-party cookies are created by domains other than the one you are visiting directly, hence the name third-party. They are used for cross-site tracking, retargeting and ad-serving.*

*Just in case you were wondering, the existence of second-party cookies is a subject of contention. Second-party cookies are cookies that are transferred from one company (the one that created first-party cookies) to another company via some sort of data partnership. For example, an airline could sell its first-party cookies (and other first-party data such as names, email addresses, etc.) to a trusted hotel chain to use for ad targeting, which would mean the cookies become classed as second-party*

**How would you manage Web Services API versioning?**

*This is a good and a tricky question. The topic of URI design is at the same time the most prominent part of a REST API and, therefore, a potentially long-term commitment towards the users of that API.*

*Since evolution of an application and, to a lesser extent, its API is a fact of life and that it's even similar to the evolution of a seemingly complex product like a programming language, the URI design should have less natural constraints and it should be preserved over time. The longer the application's and API's lifespan, the greater the commitment to the users of the application and API.*

*On the other hand, another fact of life is that it is hard to foresee all the resources and their aspects that would be consumed through the API. Luckily, it is not necessary to design the entire API which will be used until Apocalypse. It is sufficient to correctly define all the resource end-points and the addressing scheme of every resource and resource instance.*

*Over time you may need to add new resources and new attributes to each particular resource, but the method that API users follow to access a particular resources should not change once a resource addressing scheme becomes public and therefore final.*

*This method applies to HTTP verb semantics (e.g. PUT should always update/replace) and HTTP status codes that are supported in earlier API versions (they should continue to work so that API clients that have worked without human intervention should be able to continue to work like that).*

*Furthermore, since embedding of API version into the URI would disrupt the concept of hypermedia as the engine of application state (stated in Roy T. Fieldings PhD dissertation) by having a resource address/URI that would change over time, I would conclude that API versions should not be kept in resource URIs for a long time meaning that resource URIs that API users can depend on should be permalinks.*

*https://stackoverflow.com/questions/389169/best-practices-for-api-versioning*

**From a Back End perspective, are there any disadvantages or drawbacks on the adoption of Single Page Applications?**

*Let's look at one of the most popular SPA sites, GMail.*

*1. SPA is extremely good for very responsive sites:*

*Server-side rendering is not as hard as it used to be with simple techniques like keeping a #hash in the URL, or more recently HTML5 pushState. With this approach the exact state of the web app is embedded in the page URL. As in GMail every time you open a mail a special hash tag is added to the URL. If copied and pasted to other browser window can open the exact same mail (provided they can authenticate). This approach maps directly to a more traditional query string, the difference is merely in the execution. With HTML5 pushState() you can eliminate the #hash and use completely classic URLs which can resolve on the server on the first request and then load via ajax on subsequent requests.*

*2. With SPA we don't need to use extra queries to the server to download pages.*

*The number of pages user downloads during visit to my web site?? really how many mails some reads when he/she opens his/her mail account. I read >50 at one go. now the structure of the mails is almost the same. if you will use a server side rendering scheme the server would then render it on every request(typical case). - security concern - you should/ should not keep separate pages for the admins/login that entirely depends upon the structure of you site take paytm.com for example also making a web site SPA does not mean that you open all the endpoints for all the users I mean I use forms auth with my spa web site. - in the probably most used SPA framework Angular JS the dev can load the entire html temple from the web site so that can be done depending on the users authentication level. pre loading html for all the auth types isn't SPA.*

[*https://stackoverflow.com/questions/21862054/single-page-application-advantages-and-disadvantages*](https://stackoverflow.com/questions/21862054/single-page-application-advantages-and-disadvantages)

**Why do we usually put so much effort for having stateless services? What's so good in stateless code and why and when statefulness is bad?**

Because web services are based on HTTP, which is a stateless protocol. Quoting wikipedia : A stateless server is a server that treats each request as an independent transaction that is unrelated to any previous request.

*Here are some pros and cons of stateless nodes compared to stateful nodes:*

*Pros:*

*1) Stateless compute nodes consume less power and produce less heat since they do not have local hard drive.*

*2) Stateless compute nodes' root file system reside in RAM. IBM Platform HPC doesn't put root file system on NFS share for stateless compute nodes. As a result, stateless nodes in general are faster than stateful nodes.*

*3) There is no swap on stateless compute nodes.*

*4) Stateless compute nodes provision is faster than stateful nodes.*

*5) If configuration is changed on stateless nodes, a reboot will reset everything back to the original setting. For stateful nodes, if you want to revert back to the original configuration, you must reinstall the nodes.*

*Cons:*

*InfiniBand OFED support for stateless node is harder than stateful: usually the stateless image doesn't include the support for InfiniBand card, there is an extra step to put the RPMs to the installation image.*

**REST and SOAP: when would you choose one, and when the other?**

*8*

*The difference between REST and SOAP is fundamental, yet they're not that dissimilar. Ultimately, you still need to transfer exactly the same information in order to perform a particular abstract operation. It's entirely easy to make REST rather low-performing by choosing poorly what information to return, and SOAP with MTOM can transfer large binary chunks efficiently. There's even the possibility to use non-XML encodings and connected transports (e.g., XMPP) with SOAP that can make it more efficient than REST.*

*So don't worry about that!*

*A much more relevant thing to think about is that SOAP continues to have significantly more advanced tooling support in some languages, and that other languages strongly prefer REST. For example, if you want a Java client for your service, you'll be able to get going with SOAP in minutes: just put the WSDL location through a tooling engine and you've got yourself a basic client. On the other hand, if you're working with a Javascript client then you'll absolutely prefer to deal with the REST interface; Javascript works great with REST.*

*A key thing to note here is that you can have your service support both SOAP and REST at once (you might need to put them on different endpoints, but that's not very onerous). I do this with a service I support (using Java and Apache CXF) and the overhead of doing both is minimal: the key is that I need a clean abstract interface behind the scenes that both the SOAP and REST interfaces use.*

**In Web development, Model-View Controller and Model-View-View-Mode**

**l approaches are very common, both in the Back End and in the Front End. What are they, and why are they advisable?**

*It processes all the business logic and incoming requests, manipulate data using the Model component, and interact with the Views to render the final output. It receives input and initiates a response by making calls on model objects. 1) Faster development process: MVC supports rapid and parallel development.*