Eileen Pangu Following

Dec 20, 2019 · 6 min read · ✦ · Listen

System Design Interview: URL Shortener

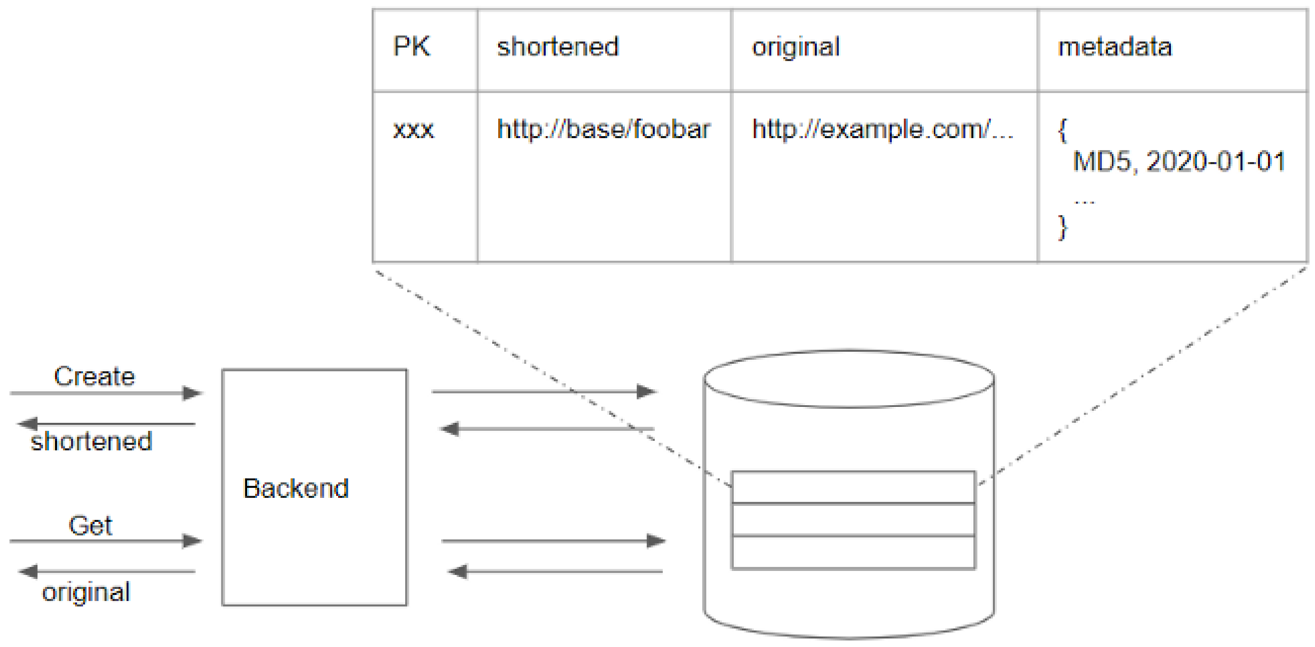
Design a URL shortener service like tinyurl.com. It is one of the most commonly asked system design interview questions. There are numerous resources online. My design is by no means the best or most complete. But I think I can offer my thought process from a different angle. After all, there is no one right answer to a system design question. It's all about focus, tradeoff, and preference. Let's dive right in.

Requirements

For any system design, as always, let's clarify who are the users and what they are trying to accomplish with the system. Let's suppose the user is a typical internet user who has a super long and crappy URL, and wants to store or share a much shorter and cleaner version of it. Now she needs a service (probably a website) through which she can create an association between the long URL and its shortened version. For simplicity, we'll assume that besides registering the association, the only other function of this service is to return an HTTP Redirect to the original URL when receiving a Get request of the shortened one.

A Simplistic Design

OK, this sounds simple enough. Let's map out the technicality behind that user journey. The service has a webpage that allows users to input the original URL in a text box, and a submit button, when clicked, sends the original URL via an HTTP Post to shortened URL. The backend stores the <shortened, original> tuple in the database. There should be an index on the shortened for efficient lookup. We don't want to use the shortened directly as the primary table key as its length may change as we update the hashing/sanitization method. So it's better not to tie ourselves to any unnecessary constraints. The backend may choose to store some metadata along the tuple, such as the hashing method, creation timestamp, etc.. Now when the backend receives a Get request, it looks up the corresponding original URL in the database and returns an HTTP redirect if one exists and a 404 if it not found. See an overview in figure 1.



PK	shortened	original	metadata
xxx	http://base/foobar	http://example.com/...	{ MD5, 2020-01-01 ... }

Figure 1

The system can be realized in various technology stacks. Many companies and teams have their own preferred ways of developing and deploying such web service. So I won't go into that detail. Let's also assume that basic operational and security safeguards exist in the stack, such as logging, monitoring and DDoS attack defense.

Problems of the Design and Mitigation

problem is from a privacy standpoint. Malicious users can scan the shortened URL space to see if they get any interesting original URLs from the system. Because think about it, the original URLs are long typically because they contain lots of query parameters, which may reveal personal information that users would hope to keep a little bit more private.

To address the first problem, we can employ those robot detection plugins on the webpage. But that limits the interaction to webpage only. For legitimate machine clients that want to use the service, we may have to introduce some extra verification mechanism such as service account. To address the second problem, we can increase the max length of returned short URLs to increase the sparsity of valid URLs inside the shortened URL space.

Another way to mitigate the two problems is to add an account system, which has lots of other benefits. The service can then distinguish different users and maybe apply different quotas. The service can also do some smart things now that it has user profiles. It can list all the shortened URLs a given user has created. It can apply permission control of accessing the original URLs, which largely resolves the privacy concern above. But a complication then comes when users want to share the shortened URLs with others. The service may allow users to mark an original URL as public for easy sharing, but more complicated access control requires non-trivial design efforts that we don’t have time to address here.

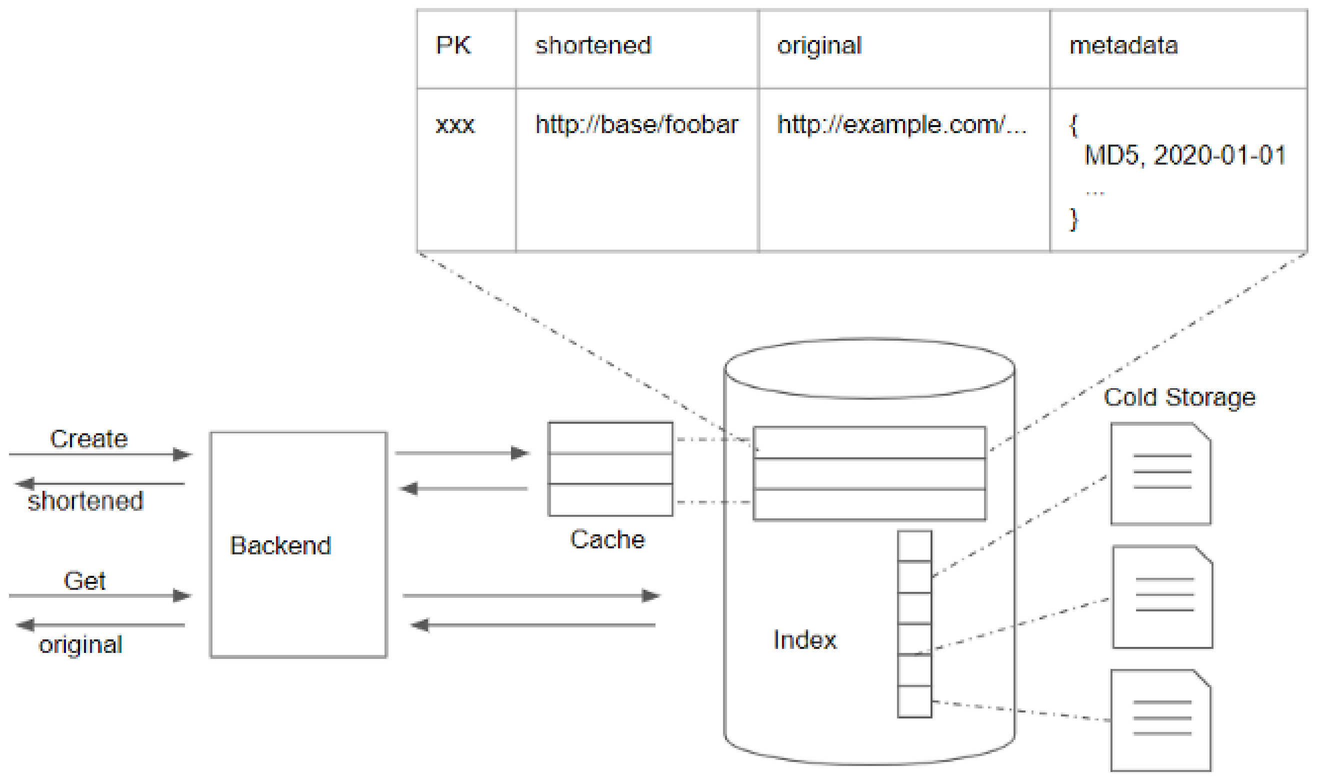
If we decide to add the account system, we’d probably implement the sign-in using some federated sign-in such as Facebook, Google, Twitter, etc. so that we only have to keep some basic account information without having to manage the password and all its accompanying rabbit holes.

> *Optimization*

OK, now our URL shortener seems functionally usable. Let’s add some performance optimization. In this case, the first thing that comes to mind is caching. Some URLs are

LRU cache would do. The cache can be realized as in memory cache or services like memcache.

Another optimization along the line of caching is hot/cold storage. This is also potentially a cost optimization. Since our service keeps accumulating entries and never deletes them, it's necessary to think about separating the frequently used entries v.s. rarely used entries, and put them in hot and cold storage respectively. One thing to call out for the cold storage is that entries inside may still be accessed one day and our service still wants to maintain reasonable SLO for them. So it's critical to build some index over the archive. We may store the index in our database and cold entries in files. Upon request, the index allows the service to identify the right file and seek for the cold entries. See figure 2 for illustration.



PK	shortened	original	metadata
xxx	http://base/foobar	http://example.com/...	{ MD5, 2020-01-01 ... }

Figure 2

We may regularly dump the cold entries from the database into files and move back to the database the ones that end up getting "warmer" in the files. Obviously, this means async process that takes the visit count in the cache and flush that into the database. If we lose some visit count due to the volatile nature of the cache, that's totally fine. For cold entries in the archive, there is no need to update their visit count. We can simply move them back into the database when they are visited. Then they enter the normal cycle of visit count updates in the database until they become too "cold" to stay.

Another feature to consider is URL expiration. It's mainly to address the forever growing database entry problem. From a technical standpoint, it's straightforward to implement — just periodically scan for and delete old entries outside a TTL. From a product standpoint, however, I am less convinced of an expiration feature. The shortened URL has wide-ranging use cases. Some may be used for a temporary link share, others may be published in papers and therefore need to stay much longer. There is no one-size-fits-all TTL. If we try to introduce a variable TTL, that'll make the system more complex for users. If users' shortened URLs become unexpectedly obsolete, they'll be mad at us regardless. So I'd rather pay the cold storage cost and offer permanent durability guarantee.

Closing

This is probably where I'd call an end to answering a system design interview question. I am sure there are a lot more to consider for building a production URL shortener. One thing we severely glossed over was the technical stack architecture. I am interested in learning whatever feedback you folks have. But I think I need to take a break until my next design question now.