Tiny URL

Shortened URL = 1/3 of the size of the actual URL

**You should always clarify requirements at the beginning of the interview and should ask questions to find the exact scope of the system that the interviewer has in mind**

Our URL shortener system should meet:

Functional requirements:

1. given a URL, our service should generate a shorter and unqieu alias of it
2. when users access a shorter URL, our service should redirect them to the original link
3. Users should optionally be able to pick a custom alias for their URL
4. Links will expire after a specific timespan automatically; users should also be able to specify expiration time.

Non-functional requirements:

1. The system should be highly available. This is required because if our service is down, all the URL redirections will start failing
2. URL redirection should happen in real-time with minimum latency
3. Shortened linked should not be guessable (not predictable).

Extended requirements:

1. Analytics, .e.g, how many times a redirection happened?
2. Our service should also be accessible through REST APIs by other services.

3. Capacity Estimation and Constraints

Our system would be **read-heavy**; there would be lots of redirection requests compared to new URL shortenings. Let’s assume 100:1 ratio between read and write.

**Traffic estimates**: if we assume that we would have 500M new URLs shortenings per month, we can expect (100 \* 500M => 50B) redirections during the same time. What would be queries per second (QPS) for our system?

New URLs shortenings per second:

~= 200 urls per second

Storage estimates: Since we expect to have 500M new URLs every month and if we would be keeping these objects for five years; total number of objects we will be storing would be 30 billion.

💡     ***Once we've finalized the requirements, it's always a good idea to define the system APIs. This would explicitly state what is expected from the system.***

API for creating and deleting URL

creatURL(api\_dev\_key, original\_url, custom\_alias=None, user\_name=None, expire\_date=None)

returns: A successful insertion returns the shortened URL, otherwise, returns an error code.

deleteURL(api\_dev\_key, url\_key)

**How do we detect and prevent abuse?**

For instance, any service can put us out of business by consuming all our keys in the current design. To prevent abuse, **we can limit users through their api\_dev\_key**, how many URL they can create or access in a certain time.

Database Design

***Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards the data partitioning.***

A few observations about nature of the data we are going to store:

1. we need to store billions of records
2. Each object we are going to store is small (less than 1K)
3. There are no relationships between records, except if we want to store which user created what URL.
4. Our service is read-heavy.

Database Schema:

We would need two tables, one for storing information about the URL mapping and the other for users’ data.

What kind of database should we use? Since we are likely going to store billions of rows and we don’t need to use relationships between objects – a NoSQL key-value store like Dynamo or Cassandra is between choice, which would also be easier to scale.

NoSQL ; cannot store UserID in the URL table (as there are not foreign keys in NoSQL), for that we would need a third table which will store the mapping between URL and the user user.

1. System design and algorithm
2. Encoding actual URL

* a unique hash (e.g. MD5 or SHA256, etc.) of the given URL. The hash can then be encoded for displaying. This encoding could be base36 ([a-z, 0-9]) or base62 ([A-Z, a-z, 0-9]) and if we add ‘-‘ and ‘.’, we can use **base64** encoding. A reasonable question would be; what should be the length of the short key? 6, 8 or 10 characters?
* Using base64 encoding, a 6 letter long key would result in 64^6 ~= 68.7 billion possible strings

Using base64 encoding, an 8 letter long key would result in 64^8 ~= 281 trillion possible strings

With 68.7B unique string, let’s assume for our system six letters keys would suffice.

MD5 algorithm 🡪 128 bit hash value.

After base64 encoding => more than 20 characters => choose 6 (or 8) letters for the key.

Issues:

1. If multiple users enter the same URL, they can get the **same shortened URL**, which is not acceptable.
2. What if parts of the URL are URL-encoded?

Workaround:

We can append an **increasing sequence number** to each input URL **to make it unique** and then generate a hash of it. We don’t need to store this sequence number in the databases, though. Possible problems with this approach could be **overflow of sequence number**? Appending an increasing sequence number will impact the performance of the service too.

Another solution could be to append user id (which should be unique) to the input URL. However, if the user has not sign in, we can ask the user to choose a unique key. Even after this, if we have a conflict, we have to keep generating a key until we get a unique one.

* Solutions: 1. use (long\_url + timestamp) as the hash function key. 2. When conflicts, regenerates the hash value(it's different because timestamp changes).

Overall, when urls are over 1 billion, there would be a lot of conflicts and the efficiency could be very low.

We can also use auto\_increment\_id in SQL database.

B. Generating key offline

We can have a standalone key generation service that generates random six letter strings before hand and stores them in a database (let’s call it key-DB). Whenever we want to shorten a URL, we will just take one of the already generated keys and use it. This approach will make things quite simple and fast since we will not be encoding the URL or worrying about duplications or collisions. KGS will make sure all the keys inserted in key-DB are unique.