LEARN TEACH

Designing a URL Shortening service like TinyURL

Let's design a URL shortening service like TinyURL. This service will provide short aliases redirecting to long URLs

Similar services: bit.ly, goo.gl, 2020.ly etc

Difficulty Level: Easy

1. Why do we need URL shortening?

URL shortening is used to create shorter aliases for long URLs. Users are redirected to the original URL when they hit these aliases. A shorter version of long URL would save a lot of space whenever we use it e.g., when printing or tweeting as tweets have a character limit.

For example, if we shorten this page through TinyURL:

LEARN TEACH

https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR0ZNZ5lpwb9l9MzRkOz9Pilq8/shortener-system/5jEMzmqZ6Z6

We would get:

LEARN TEACH

http://tinyurl.com/jlg8zpc

The shortened URL is nearly 1/3rd of the size of the actual URL.

URL shortening is used for optimizing links across devices, tracking individual links to analyze audience and campaign performance, and hiding affiliated original URLs, etc.

If you haven't used tinyurl.com before, please try creating a new shorten URL and spend some time going through different options their service offers. This will help you a lot in understanding this chapter better.

2. Requirements and Goals of the System

LEARN TEACH

the exact scope of the system that the interviewer has in mind.

Our URL shortener system should meet the following requirements:

Functional Requirements:

1. Given a URL, our service should generate a shorter and unique 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 important 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 links should not be guessable (not predictable).

Extended Requirements:

LEARN TEACH

the exact scope of the system that the interviewer has in mind.

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 100M 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:

100 million / (30 days * 24 hours * 3600 seconds) = ~40 URLs/s

URLs redirections per second:

50 billion / (30 days) * 24 hours * 3600 sec) = ~19K/s

Storage estimates: Since we expect to have 100M 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.

LEARN TEACH

Let's assume that each object we are storing can be of 500 bytes (just a ballpark, we will dig into it later); we would need 15TB of total storage:

30 billion * 500 bytes = 15 TB

URL Shortenings per month	100	million
Total years	5	
URL object size	500	Bytes
Total Files	30	billion
Total Storage	15	TB

LEARN TEACH

and would be 100GB per second.

100 * 500 Bytes = 100 KB/s

For read requests, since every second we expect ~19K URL redirections, assuming one byte data for our service would be 19GB per second.

19K * 500 Bytes = ~9 MB/s

Memory estimates: If we want to cache some of the hot URLs that are frequently accessed, how much memory would we need to store them? If we follow the 80-20 rule, meaning 20% of URLs generating 80% of traffic, we would like to cache these 20% hot URLs.

Since we have 19K requests per second, we would be getting 1.7billion requests per day.

19K * 3600 seconds * 24 hours = ~1.7 billion

To cache 20% of these requests, we would need 170GB of memory.

0.2 * 1.7 billion * 500 bytes = ~170GB

LEARN TEACH

For more estimation and back-of-the-envelope calculations, you can refer to this post.

New URLs	200/s
URL redirections	19K/s
Incoming data	100KB/s
Outgoing data	9MB/s
Storage for 5 years	15TB
Memory for cache	170GB

LEARN TEACH

4. System APIs

LEARN TEACH

Once we've finalized the requirements, it's always a good idea to define the system APIs. This would

LEARN TEACH

creating and deleting URLs:

```
create(api_dev_key, original_url, custom_alias=None user_name=None, expire_date=None)
```

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.

original_url (string): Original URL to be shortened.

custom_alias (string): Optional custom key for the URL.

user_name (string): Optional user name to be used in encoding.

expire_date (string): Optional expiration date for the shortened URL.

Returns: (string)

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

```
deleteURL(api_dev_key, url_key)
```

Where "url_key" is a string representing the shortened URL to be retrieved. A successful deletion returns 'URL Removed'.

LEARN TEACH

access in a certain time.

5. Database Design

LEARN TEACH

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 the nature of 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 5K).
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 mappings and the other for user data.

LEARN TEACH

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 a better choice, which would also be easier to scale. Please see SQL vs NoSQL for more details. The rest of this section assumes we are using a NoSQL store like Dynamo or Cassandra.

Using hashed encoding, a 6 letter long key would result in 64^6 = ~68.7 billion possible strings.

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

With 68.7B unique strings, let's assume that our system an letters keys would suffice.

If we use the MD5 algorithm as our hash function, it'll produce a 128-bit hash value. After base64 encoding we'll get a string having at least 22 characters, how will we choose our key from this? We can take the first 6 (or 8) letters for the key. This could result in key duplication though, upon which we can choose some other characters out of the encoding string or swap some characters.

What are different issues with our solution? We have the following couple of problems with our encoding scheme:

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? e.g., http://www.educative.io/distribute/ask.php?id=design, and http://www.educative.io/distribute/ask.php?id=%3Adesign are identical except for the URL encoding.

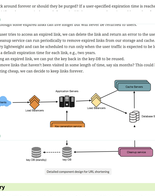
Workaround for the issues: We can append an incremental sequence number to each input URL to make it unique and then

LEARN TEACH

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 signed in, we can ask the user to choose a uniqueness key. Even after this if we have a conflict, we have to keep generating a key until we get a unique one.

LEARN TEACH



6. Generating keys offline

We can have a standalone Key Generation Service (KGS) that generates random six letter strings beforehand 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 to key-DB are unique.

Can concurrency cause problems? As soon as a key is used, it should be marked in the database so that it doesn't get used again. If there are multiple servers reading keys concurrently, we might get a scenario where two or more servers try to read the same key from the database. How can we solve this concurrency problem?

Servers can use KGS to read/mark keys in the database. KGS can use two tables to store keys, one for keys that are not used yet and one for all the used keys. As soon as KGS gives keys to one of the servers, it can move them to the used keys table. KGS can always keep some keys in memory so that whenever a server needs them, it can quickly provide them. For

LEARN TEACH

servers. For that, it must synchronize (or get a lock) on the data structure holding the keys before removing keys from it and giving them to a server.

What would be the key-DB size? With base64 encoding, we can generate 68.7B unique six letters keys. If we need one byte to store one alpha-numeric character, we can store all these keys:

6 (characters per key) * 68.7B (unique keys) = 412 GB.

Isn't KGS the single point of failure? Yes, it is. To solve this, we can have a standby replica of KGS, and whenever the primary server dies, it can swap over to generate and provide keys.

Can each app server cache some keys from key-DB? Yes, this can surely speed things up. Although in this case, if the application server dies before consuming all the keys, we will end up losing those keys. This could be an acceptable since we have 68B unique six letters keys.

How would we handle concurrent reads? We can have a lock or mutex (or any other synchronization mechanism) on the read

LEARN TEACH

LEARN TEACH

9. Load Balancer (LB)

We can add Load balancing layer at three places in our system:

1. Between Clients and Application servers
2. Between Application servers and database servers
3. Between Application Servers and Cache servers

Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is if a server is dead, LB will take it out of the pool and will stop sending any traffic to it. A problem with Round Robin LB is, it won't take server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. In a MySQL Master-Slave setup, Round Robin LB solution can be placed that periodically queries backend server about its load and adjusts traffic based on that.

10. Purging or DB cleanup

Should entries stick around forever or should they be purged? If a user-specified expiration time is reached, what should

LEARN TEACH

Here are potential strategies regarding when Cache servers evicting cache entries:

- Whenever a user tries to access an expired link, we can delete the link and return an error to the user.
- A separate Cleanup service can run periodically to remove expired links from our cache and storage. This service should be very lightweight and can be scheduled to run only when the user traffic is expected to be low.
- We can have a default expiration time for each link, e.g., two years.
- After removing an expired link, we can put the key back in the key-DB to be reused.
- Should we remove links that haven't been visited in some length of time, say six months? This could be tricky. Since storage is getting cheap, we can decide to keep links forever.

LEARN TEACH

11. Telemetry

How many times a short URL has been used, what were user locations, etc.? How would we store these statistics? If it is part of a DB row that gets updated on each view, what will happen when a popular URL is shortened with a large number of concurrent requests?

We can have another about the country of the visitor, date and time of access, web page that refers the click, browser or platform from where the page was accessed and more.

12. Security and Permissions

LEARN TEACH

an error (HTTP 401) back. Given that, we are storing our data in a NoSQL wide-column database like Cassandra, the key for the table storing permissions would be the 'Hash' (or the MD5 generated hash), and the column will store the UserIDs of those users that have permissions to see the URL.

✓ Completed Next

Discussion

Type a discussion topic

Sequencing Numbers

System Design Interviews: A step by...

Send feedback or ask a question 0 Recommendations