8. Design a URL Shortener

Alex Xu - System Design Interview (2nd ed.)

Step 1. Understand the problem and establish design scope

Requirements

100 million URLs are generated per day

Shortened URL can be a combination of numbers (0-9) and characters (a-z, A- Z)

Shortened URLs cannot be deleted or updated

Back of the envelope estimation

⚠ Write ops: 100 million / 24 /3600 = 1,160 per second!

⚠ Read ops: assume x10 => 11,600 per second

Assume service should run for 10 years: 100 million * 365 * 10 = 365 billion records

Assume average URL length is 100

⚠ Storage requirement over 10 years: 365 billion * 100 bytes * 10 years = 365 TB

Step 2. Propose high-level design and get buy-in

API Endpoints

POST api/v1/data/shorten

request parameter: {longUrl: longURLString}

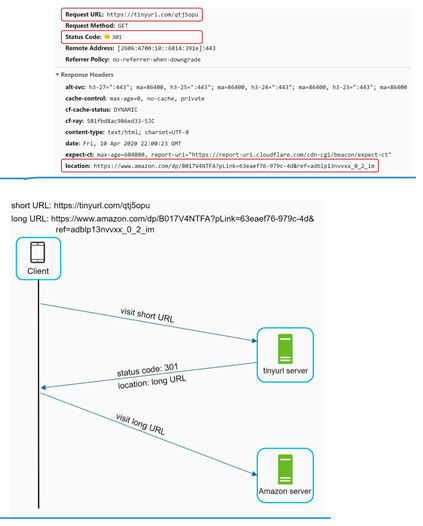
return shortURL

GET api/v1/shortUrl

Return longURL for HTTP redirection

URL redirecting

Once the server receives a tinyurl request, it changes the short URL to the long URL with 301 redirect



The detailed communication between clients and servers

301 redirect vs 302 redirect

301 redirect is permanent - cached by browser

Good when want to reduce the load on server

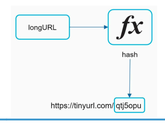
302 redirect is temporary

Good if analytics is important

Implementation via hash tables: shortURL -> longURL

URL shortening

www.tinyurl.com/{hashValue}



Each longURL must be hashed to one hashValue

Each hashValue can be mapped back to the longURL

Step 3. Design deep dive

Data model

1. We start from in-memory hash-table

But this is not feasible for real world scenarios

url Table	
PK	**id (auto increment)**
	shortURL
	longURL

Q: why do we need id with auto increment? It will be another big issue here considering high load - we will need ID generator from previous chapter!

2. Better option is to use relational database table

Q: why use relational(!) database? they are less scalable

Hash function

hashValue

consists of characters from [0-9, a-z, A-Z]

containing 10 + 26 + 26 = 62 possible chars

find the smallest n such that 62^n ≥ 365 billion

n=7

Two types of hash function

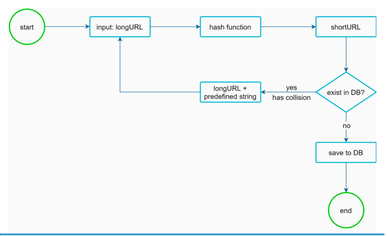
Hash function	Hash value (Hexadecimal)
CRC32	5cb54054
MD5	5a62509a84df9ee03fe1230b9df8b84e
SHA-1	0eeae7916c06853901d9ccbefbfcaf4de57ed85b

1. Hash + collision resolution

Well-known hash functions

But even the shortest version is too long (> 7 chars)

If we take first 7 chars then there will be more collisions



To resolve hash collisions, we can recursively append a new predefined string until no more collision is discovered

But it is expensive to query the database to check if a shortURL exists for every request

Bloom filters can improve performance

We prefer this approach!

That's why we need ID!

2. Base 62 conversion

Just convert auto-increment ID to 62-bit

Q: But we don't need shortURL now - as it is a duplication of ID?!

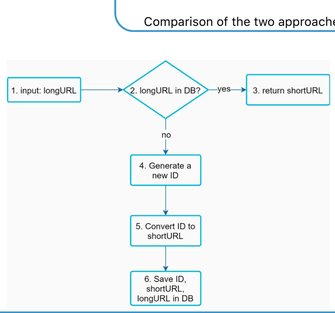
Q: However it will allow users to generate multiple short URLs for the same longURL???

Yes - we should now check if longURL already exist in our DB

Q: All shortURLs can now be iterated!? Yes. This can be a security concern!

Hash + collision resolution	Base 62 conversion
Fixed short URL length.	The short URL length is not fixed. It goes up with the ID.
It does not need a unique ID generator.	This option depends on a unique ID generator.
Collision is possible and must be resolved.	Collision is impossible because ID is unique.
It is impossible to figure out the next available short URL because it does not depend on ID.	It is easy to figure out the next available short URL if ID increments by 1 for a new entry. This can be a security concern.

Comparison of the two approaches



URL shortening deep dive

The distributed unique ID generator (from prev chapter!) is worth mentioning!

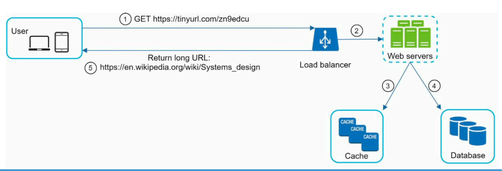
Q: This will also solve security concern!?

Q: How to check longURL? How DB table should be indexed and distributed (partitioned)?

If we partition data by longURL (to quickly check it) then how can we retrieve data by ID/shortURL?

If we partition data by ID/shortURL then how can we quickly check longURL?

URL redirecting deep dive



Step 4. Wrap up

additional considerations

Rate limiter

Web server scaling

Since the web tier is stateless, it is easy to scale the web tier by adding or removing web servers

Database scaling

Database replication and sharding are common techniques

Analytics

Availability, consistency, and reliability