System Design: Tiny URL

1. Requirements

1. Functional Requirements

1. Given a URL, our service should generate a shorter and unique alias of it. This is called short link. This short link should be short enough to be copied and pasted into applications.
2. When users access short link, our service should be able to redirect them to original link.
3. Users should be able to pick up a custom short link for their URL.
4. Links will expire after a standard default timespan. Users should be able to specify the expiration time.

2. Non-functional Requirements

1. The system should be highly available. This is required because if the service is down, all the URL redirections will start failing.
2. URL redirection should happen real-time with minimum latency.
3. Short links should not be predictable.

3. Extended Requirements

1. Analytics: How many times redirection happened
2. Our service should be available through REST APIs by other services.

2. Capacity Estimation and Constraints

TinyURL will be a read-heavy system, meaning there will be more read requests than write requests. Let the read : write ratio be 100 : 1.

1. Traffic estimates:

Suppose we have 1 Billion users. On average, each user makes 5 write requests.
Hence, = 1 B (users)* 5 (request/user) / 30(days) * 24 (hrs) * 60 (min) * 60 (sec)
= 2000 qps
Write: 2000 qps

We assumed read to write ratio be 100 : 1
Hence, read : 200 K qps

As there are 1B users per month, they will create 5 * 1B = 5 B queries,
Hence, per month
Write queries : 5 B
Read queries: 500 B

2. Disk Space:

Assuming that each URL takes 300 bytes and all total (username, hash, timestamp) take 500 bytes.

Per month, 1B (users) * 5 (queries per person per month) * 500 bytes (size of one query)
For 10 years, = 10^9 * 5 * 500 * 10 * 12 bytes = 10^{12} *(25 *12) = 300 * 10^{12} = 300 TB for 10 years storage. (For schema : 500 bytes)

3. Bandwidth:

Incoming data (write) : 500 bytes * 2000 qps = 10^6 bps = 1000 kbps
Outgoing data (read) : 500 bytes * 200 k qps = 10^8 bps = 100 mbps

4. Memory:

Using 80-20 principle, (20% users use 80% space),
Reading, we have 2000 qps = In a day, 200k * 24 * 60 * 60 = 1728 * 10^7 = 17.28 Billion request per day, to cache 20% of them, = 17.28 B * 0.2 * 500(size of a request) bytes = 17.28 * 100 * B = 1.728 TB/ day ~ 2 TB/day

3. System APIs

We can have SOAP / REST APIs to expose functionality of a service.
Following can be the APIs for creating and deleting short-links.

1. createURL(api_dev_key, original_URL, custom_alias, user_id, expiration_date):

Parameters:

1. api_dev_key (String) : The API key of registered account. Used to prevent DDOS attacks by throttling users based on their allocated quota.
2. original_URL (String) : Original URL to be shortened
3. custom_alias (String): Optional custom alias if requested by user
4. user_id (String): Optional, can be used to encode for generating short urls
5. expiration_date (String): Optional expiration date for shortened URL

Returns: (String) Success : short-link, else, error code

2. deleteURL(api_dev_key, short_link, user_id):

Parameters:

1. api_dev_key (String): The API key of registered account. Used to prevent DDOS attacks by throttling users based on their allocated quota.
2. Short_link (String): Short_link to be deleted
3. user_id (String): to authenticate if given short-URL belongs to the same user_id or not

Returns: (String) Success or Failure message

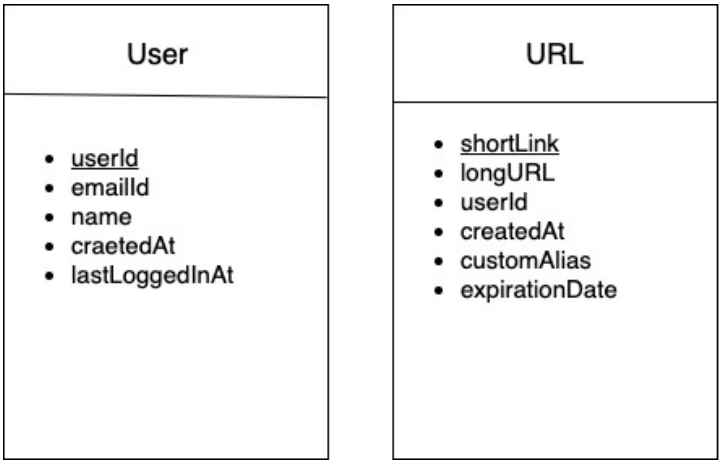
4. Database Design

After capacity estimation, we came to know that

1. Our system is read-heavy
2. There is no relationship in between the rows
3. There are billions of records

Hence, we decide to use NoSQL database like dynamoDB. Also, NoSQL databases are easier to scale

We will only have two entities : User and URL

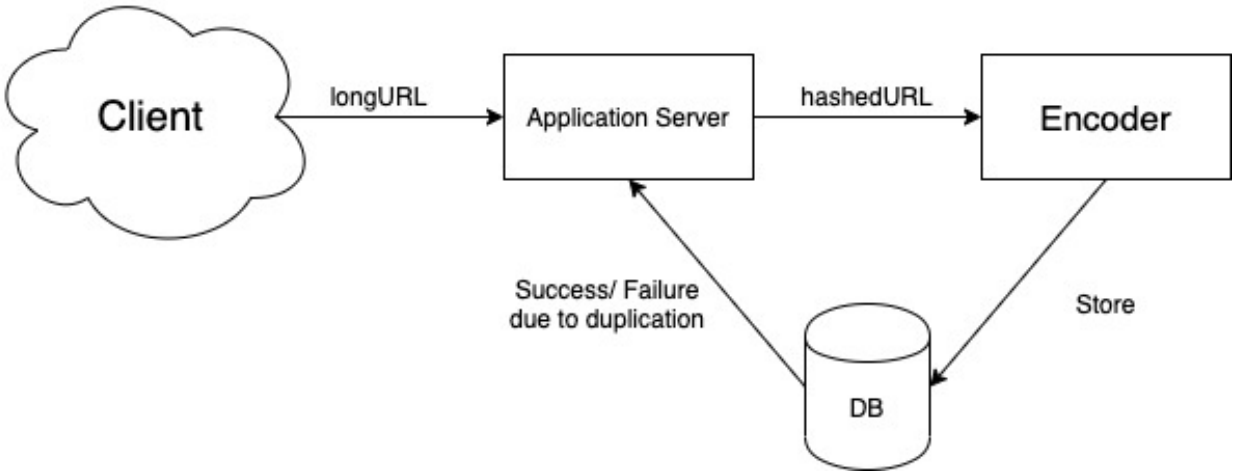


5. High Level Design

The main problem here is how to design a system which will shorten the URL.
We need to create a short-link of size 6/7/8.

1. Solution 1: Encoding actual URL

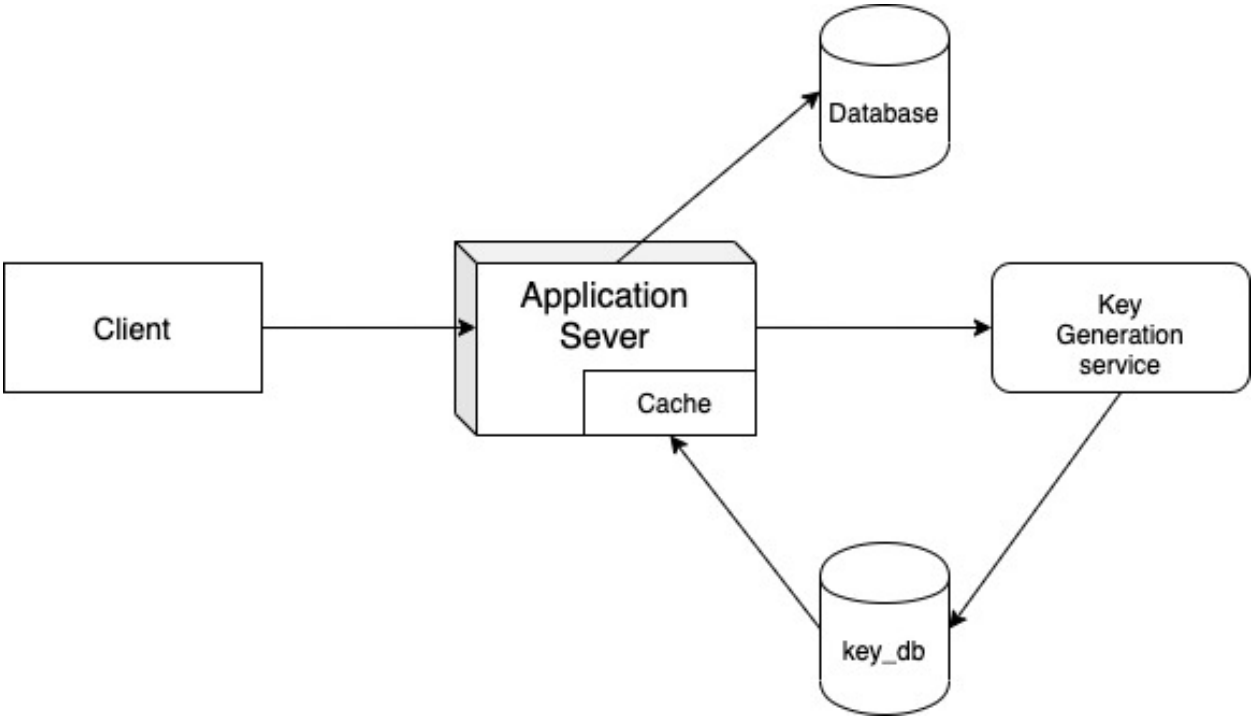
We can compute unique hash using MD5(128 bits) or SHA256 (256 bits). To shorten it even more, we can use base36 [a-z, 0-9] or base64 [a-z, A-Z, 0-9, /, _]
Using 64 bit encoding for 6 characters will create 64 ^ 6 possible strings
Using 64 bit encoding for 8 character will create 64 ^ 8 possible strings which are more than needed.
But, if we create hash using MD5 and encode it using base36, and only take first 6 characters, there can be a possible duplication of keys. The solution for this is to append an increasing sequence number. But this can also lead to overflow. Another solution is to append unique user_id to the hash code, but again, this will ask user to approve unique key. If the user is not online, this might fail. Hence, we need another technique.



2. Solution 2: Key Generation Service

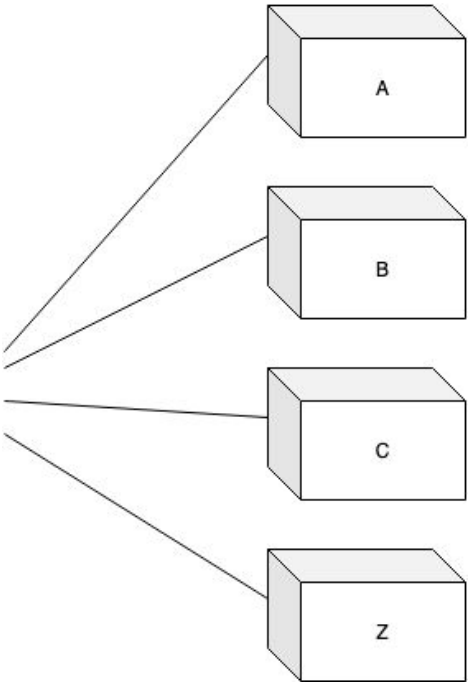
Key Generation Service (KGS) generates 6-bit short-links and stores then in a database – key_db. When we need to encode incoming longURL, we just take one from already created key_db and use it. (We mark the used key as marked).

To avoid concurrency (due to multiple requests at the same time), we can create two databases. One database will contain used keys and another one will contain unused keys. As soon as a used key is been deleted, it will be shifted to unused one. But again, to use this, we will have to call multiple times to database. To overcome this, we can store some keys in cache of our application servers and use them on go. KGS can also be a single point of failure. Hence, we need a stand-alone replica of it.



6. Low Level Design : Data Partitioning and Replication

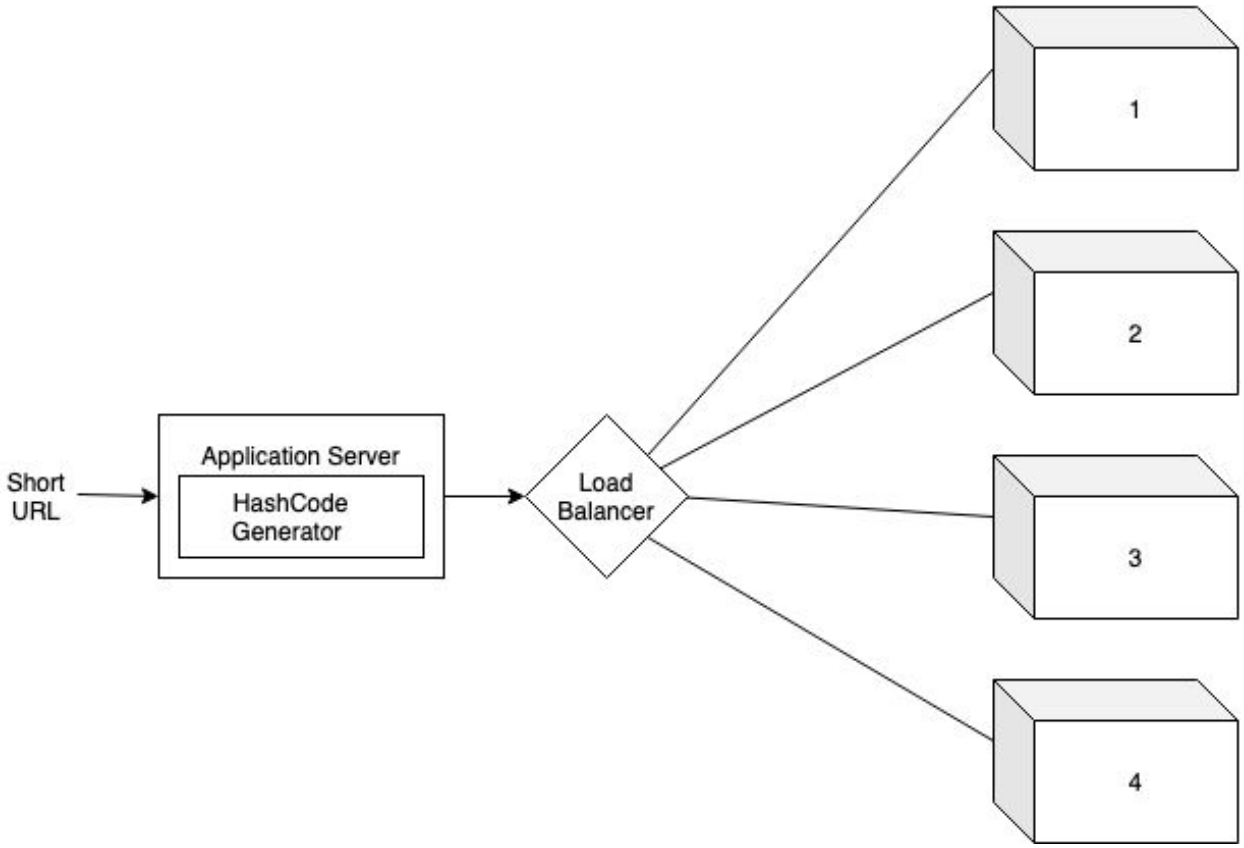
a. Range based data partitioning:
Divide database in n partitions to reduce burden on database. Find a technique to divide contents for example: starting character. Along with it, we also create multiple copies of every partition to avoid single point of failure. The problem here is unbalanced data-partitioning. Hence, we need another method.



If short URL starts from a/A, load balancer will redirect it to first partition and will return longURL from there.

b. Hash based data partitioning:
We find hash of short_link and find out the partition. For example, we have n servers and we find hash as incoming_number % n. Then we will get particular partition.

Example: shortURL : 31
4 servers
Hashcode 31%4 = 3, data is in 3rd partition.



7. Cache

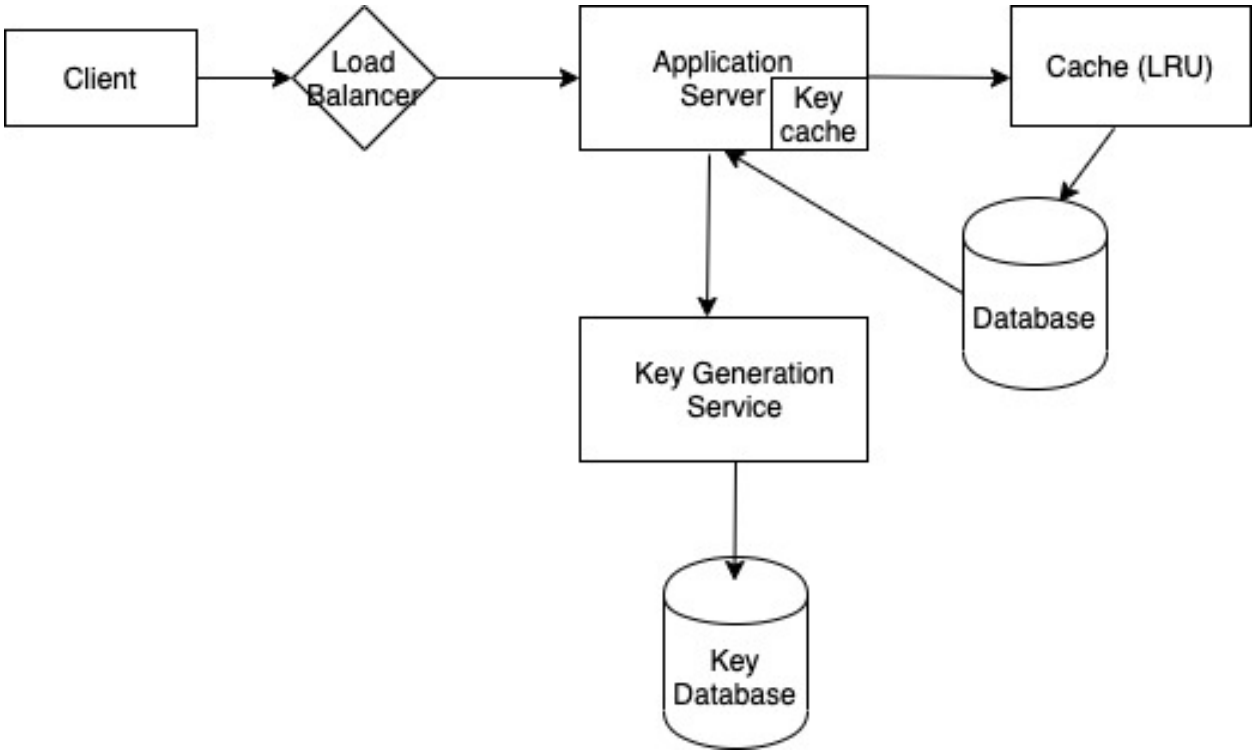
We can use LRU (Replace least recently used cache) technique in this case. We can replicate cache to avoid single point of failure. Also, we can use write_back caching for low latency.

8. Load Balancing

For load balancing, we can use least response time technique. Else, as round robin can result in uneven weight distribution (as different servers have different capacities), we can use weighted round robin technique.

9. Low Level Design

Every component is replicated to avoid single point of failure. Load balancers are expensive; hence, we use single stand-by load balancer.



10. Cleanup

Clean up can be performed by three ways

1. Default URL Expiration: Can run cron-job which will run periodically in non-business hours/ least busy hours to lessen burden on the system.
2. Lazy Cleanup: When the user hits the URL, check if expired or not. If expired, put the short-link in unused key database and send expired message to the user. The data will be removed once in say, 6 months to avoid server burden on cleanup service.

11. Telemetry

Statistics about how many times a user used a service, how many times a particular URL was requested, how many times user logged in, geographic location of users etc can be calculated using our service.

12. Security

To secure system from DDOS attacks, we can limit the number of URLs that can be created per user. We need to authenticate user before every time he uses the system. Access permissions can also be implemented upon user request.