System API

Create URL(api_key , original URL, custom alias = None, username = None, expiration date = None)

Api_key → Api key for registered developer account and can be used for maintaining quota

Original URL → URL to be shortened

Custom Alias → Optional Customer key for URL shortened

Username → Optional username to be used for encoding in shortened URL

Expiration Date → Optional expiration time for shortened URL

Delete URL(api_key, shortened URL)

Shortened URL → URL to be deleted. Successful deletion returns “URL has been deleted successfully”

Prevent and Detection of Abuse

The current system design is too basic in nature. We can run out of all the URL keys if a malicious user consumes uses all the keys. To prevent abuse, we can limit users via their api_key. Each api_key can be limited to a certain number of URL creations and redirections per some time period. However, user can create tiny URLs without login and registration into system. In such case, we can make user of URL IP Address, Mac address ,Geolocation to limit and prevent the abuse. This may require adding more information(parameters) in system api to accomplish the task.

Database Design

Nature of Data

- Require storage for billions of records
- Smaller Object Size(less than 1K)
- No relation exists between users and generated URLs.
- Service is read heavy

Database Schema

URL Table

Property	**Type**
Hashed URL	Varchar(16)
Original URL	Varchar(512)
Creation Date	Datetime
ExpirationDate	Datetime
Use rid	int
IP Address	long

User

User Id	Int(primary key)
Name	Varchar(40)

Email	Varchar(40)
Creation Date	Datetime
Last Login	datetime

Keep above nature of data, we propose to use NoSQL database such as DynamoDB, Mongo DB, Cassandras for our service.

Basic System Design and Algorithm

a.) **Encoding Original URLs**

We can compute 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 chars

Using base64 encoding, a 6 letters long key would result in 64^6 = ~68.7 billion possible strings
Using base64 encoding, an 8 letters long key would result in 64^8 = ~281 trillion possible strings

Problems during URL generation

Uniqueness of Shortened URL

Multiple users enter the same URL, might get the same shortened URL, which is not acceptable

Solution

Append the unique user id to the input URL to resolve the duplicate issues. In case user is not signed in, we can mandate user to choose a unique key.

b.) Offline Key Generation Service

We can have a standalone **Key Generation Service (KGS)** that generates random six-letter strings beforehand and stores them in a database (let us 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. Not only are we not encoding the URL, but we will not have to worry about duplications or collisions. KGS will make sure all the keys inserted into key-DB are unique

Can concurrency cause problems? As soon as a key is used, it should be marked in the database to ensure it does not get reuse. 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 it can quickly provide them whenever a server needs them.

For simplicity, as soon as KGS loads some keys in memory, it can move them to the used keys table. This ensures each server gets unique keys. If KGS dies before assigning all the loaded keys to some server, we will be wasting those keys–which could be acceptable, given the huge number of keys we have.

KGS also must make sure not to give the same key to multiple 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 in:

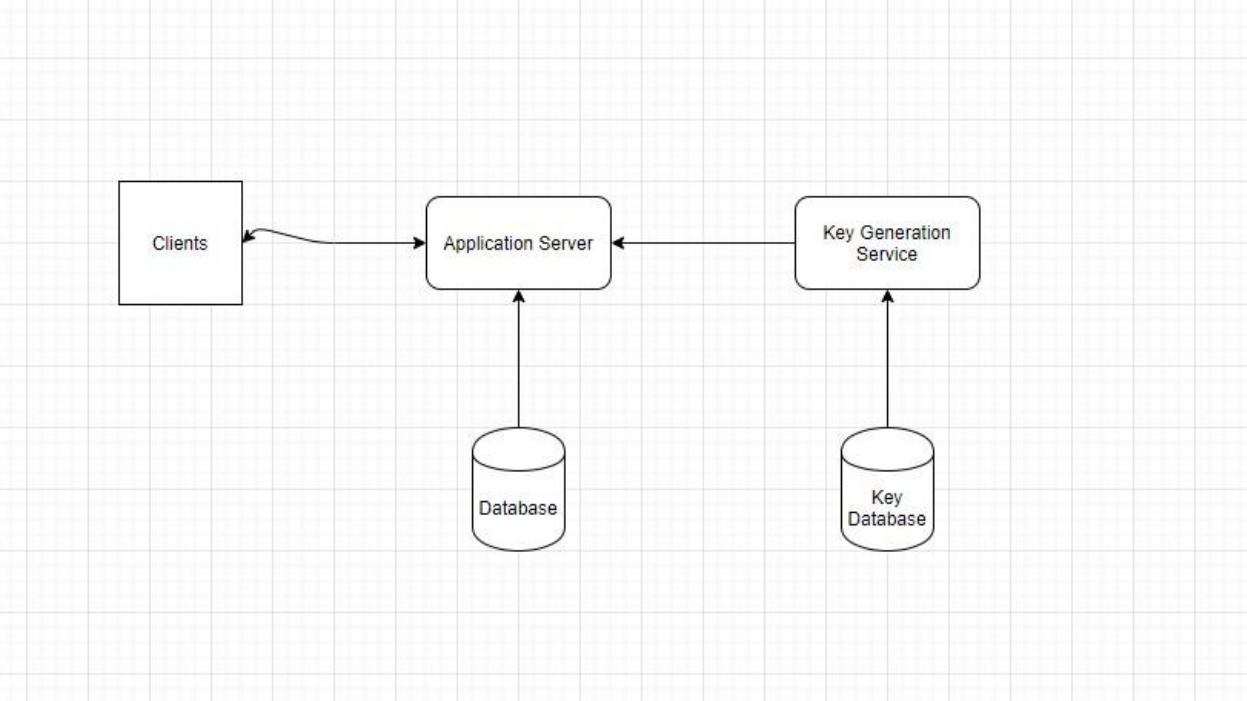
6 (characters per key) * 68.7B (unique keys) = 412 GB.

Isn't KGS a single point of failure? Yes, it is. To solve this, we can have a standby replica of KGS. Whenever the primary server dies, the standby server can take 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 can be acceptable since we have 68B unique six-letter keys.

How would we perform a key lookup? We can look up the key in our database to get the full URL. If it is presents in the DB, issue an "HTTP 302 Redirect" status back to the browser, passing the stored URL in the "Location" field of the request. If that key is not present in our system, issue an "HTTP 404 Not Found" status or redirect the user back to the homepage.

Should we impose size limits on custom aliases? Our service supports custom aliases. Users can pick any ‘key’ they like, but providing a custom alias is not mandatory. However, it is reasonable (and often desirable) to impose a size limit on a custom alias to ensure we have a consistent URL database. Let us assume users can specify a maximum of 16 characters per customer key (as reflected in the above database schema).



Data partitioning and Replication

To scale out our DB, we need to partition it so that it can store information about billions of URLs. We need to come up with a partitioning scheme that would divide and store our data into different DB servers.

a. Range Based Partitioning: We can store URLs in separate partitions based on the first letter of the hash key. Hence we save all the URLs starting with letter 'A' (and 'a') in one partition, save those that start with letter 'B' in another partition and so on. This approach is called range-based partitioning. We can even combine certain less frequently occurring letters into one database partition. We should come up with a static partitioning scheme so that we can always store/find a URL in a predictable manner.

The main problem with this approach is that it can lead to unbalanced DB servers. For example, we decide to put all URLs starting with letter 'E' into a DB partition, but later we realize that we have too many URLs that start with the letter 'E'.

b. Hash-Based Partitioning: In this scheme, we take a hash of the object we are storing. We then calculate which partition to use based upon the hash. In our case, we can take the hash of the 'key' or the short link to determine the partition in which we store the data object.

Our hashing function will randomly distribute URLs into different partitions (e.g., our hashing function can always map any 'key' to a number between [1…256]), and this number would represent the partition in which we store our object.

Cache

We can cache URLs that are frequently accessed. We can use some off-the-shelf solution like Memcached and Redis , which can store full URLs with their respective hashes. The application servers, before hitting backend storage, can quickly check if the cache has the desired URL.

How much cache memory should we have? We can start with 20% of daily traffic and, based on clients' usage pattern, we can adjust how many cache servers we need. As estimated above, we need 170GB memory to cache 20% of daily traffic. Since a modern-day server can have 256GB memory, we can easily fit all the cache into one machine. Alternatively, we can use a couple of smaller servers to store all these hot URLs.

Which cache eviction policy would best fit our needs? When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently used URL first. We can use a Linked Hash Map or a similar data structure to store our URLs and Hashes, which will also keep track of the URLs that have been accessed recently.

To further increase the efficiency, we can replicate our caching servers to distribute the load between them.

How can each cache replica be updated? Whenever there is a cache miss, our servers would be hitting a backend database. Whenever this happens, we can update the cache and pass the new entry to all the cache replicas. Each replica can update its cache by adding the new entry. If a replica already has that entry, it can simply ignore it.

Load balancer

We can add a 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, we could use a simple Round Robin approach 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 that if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it.

A problem with Round Robin LB is that we do not take the server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that.