## 1. Why do we need URL shortening?

URL shortening is used to create shorter aliases for long URLs. We call these shortened aliases “short links.” Users are redirected to the original URL when they hit these short links. Short links save a lot of space when displayed, printed, messaged, or tweeted. Additionally, users are less likely to mistype shorter URLs.

## 2. Requirements and Goals of the System

**Functional Requirements:**

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

## 3. Capacity Estimation and Constraints

Our system will be read-heavy. There will be lots of redirection requests compared to new URL shortenings. Let’s assume a 100:1 ratio between read and write.

**Traffic estimates:** Assuming, we will have 500M new URL shortenings per month, with 100:1 read/write ratio, we can expect 50B redirections during the same period:

100 * 500M => 50B

**Storage estimates:** Let’s assume we store every URL shortening request (and associated shortened link) for 5 years. Since we expect to have 500M new URLs every month, the total number of objects we expect to store will be 30 billion:

500 million * 5 years * 12 months = 30 billion

Let’s assume that each stored object will be approximately 500 bytes (just a ballpark estimate–we will dig into it later). We will need 15TB of total storage:

30 billion * 500 bytes = 15 TB

## 4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definitions of the APIs for creating and deleting URLs:

**API 1-:**

```
createURL(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 the encoding.

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

**Returns:** (string)

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

**API 2:-**

```
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’.

## 5. Database Design

A few observations about the nature of the data we will store:

1. We need to store billions of records.
2. Each object we store is small (less than 1K).
3. There are no relationships between records—other than storing which user created a URL.
4. Our service is read-heavy.

**Database Schema:**

We would need two tables: one for storing information about the URL mappings and one for the user’s data who created the short link.

**What kind of database should we use?** Since we anticipate storing billions of rows, and we don’t need to use relationships between objects – a NoSQL store like DynamoDB, Cassandra or Riak is a better choice. A NoSQL choice would also be easier to scale.

## 6. Basic System Design and Algorithm

The problem we are solving here is how to generate a short and unique key for a given URL.

In the TinyURL example in Section 1, the shortened URL is "https://tinyurl.com/rxcsyr3r". The last eight characters of this URL constitute the short key we want to generate. We'll explore two solutions here:

**a. Encoding actual URL**

We can compute a unique hash (e.g., MD5 or SHA256, etc.) of the given URL. The hash can then be encoded for display. 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 letters long key would result in 64^6 = ~68.7 billion possible strings.

With 68.7B unique strings, let's assume six letter keys would suffice for our system.

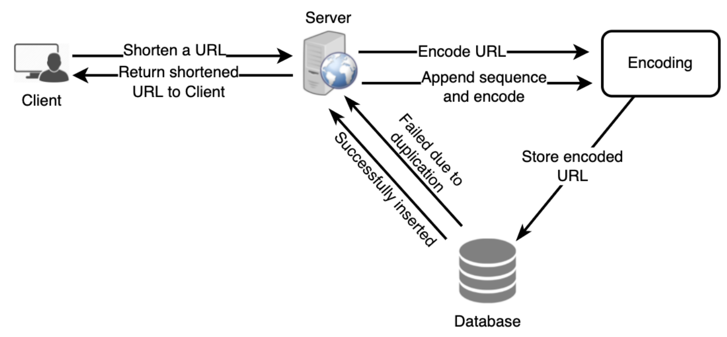
If we use the MD5 algorithm as our hash function, it will produce a 128-bit hash value. After base64 encoding, we'll get a string having more than 21 characters (since each base64 character encodes 6 bits of the hash value). Now we only have space for 6 (or 8) characters per short key; how will we choose our key then? We can take the first 6 (or 8) letters for the key. This could result in key duplication;

**What are the 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.

**Workaround for the issues:** We can append an increasing sequence number to each input URL to make it unique and then generate its hash. We don't need to store this sequence number in the databases, though. Possible problems with this approach could be an ever-increasing sequence number. Can it overflow? Appending an increasing sequence number will also impact the performance of the service.

Another solution could be to append the user id (which should be unique) to the input URL. However, if the user has not signed in, we would have to 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.



Request flow for shortening of a URL

**1** of 9

### b. 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 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 won't 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 that it is not 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 to quickly provide them whenever a server needs them.

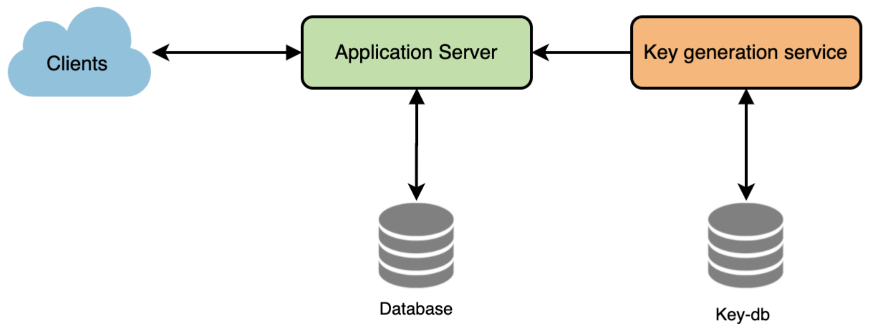
For simplicity, as soon as KGS loads some keys in memory, it can move them to the used keys table. This ensures each server gets unique keys. If KGS dies before assigning all the loaded keys to some server, we will be wasting those keys–which could be acceptable, given the huge number of keys we have.

KGS also has to 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.

**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’s present 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.



High level system design for URL shortening

## 7. Data Partitioning and Replication

To scale out our DB, we need to partition it so that it can store information about billions of URLs. Therefore, we need to develop 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 hash key's first letter. Hence we will save all the URL hash keys starting with the letter 'A' (and 'a') in one partition, save those that start with the 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. Thus, we should develop a static partitioning scheme to 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 the 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]). This number would represent the partition in which we store our object.

This approach can still lead to overloaded partitions, which can be solved using Consistent Hashing.

## 8. Cache#

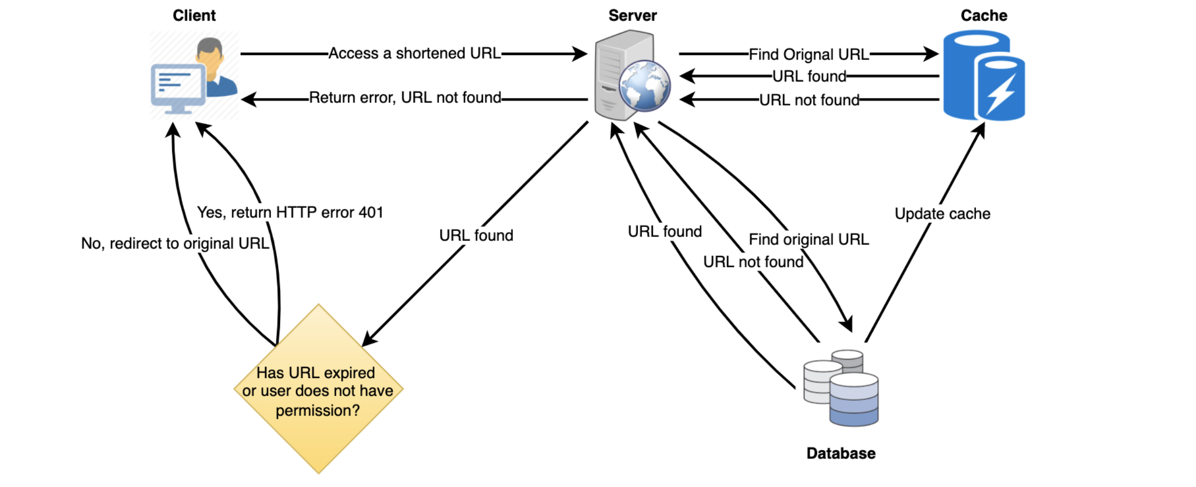
We can cache URLs that are frequently accessed. We can use any off-the-shelf solution like Memcached, which can store full URLs with their respective hashes. Thus, the application servers, before hitting the backend storage, can quickly check if the cache has the desired URL.

**How much cache memory should we have?** We can start with 20% of daily traffic and, based on clients' usage patterns, we can adjust how many cache servers we need. As estimated above, we need 170GB of memory to cache 20% of daily traffic. Since a modern-day server can have 256GB of 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.



Request flow for accessing a shortened URL

**1** of 11

## 9. Load Balancer (LB)

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 stop sending any traffic to it.

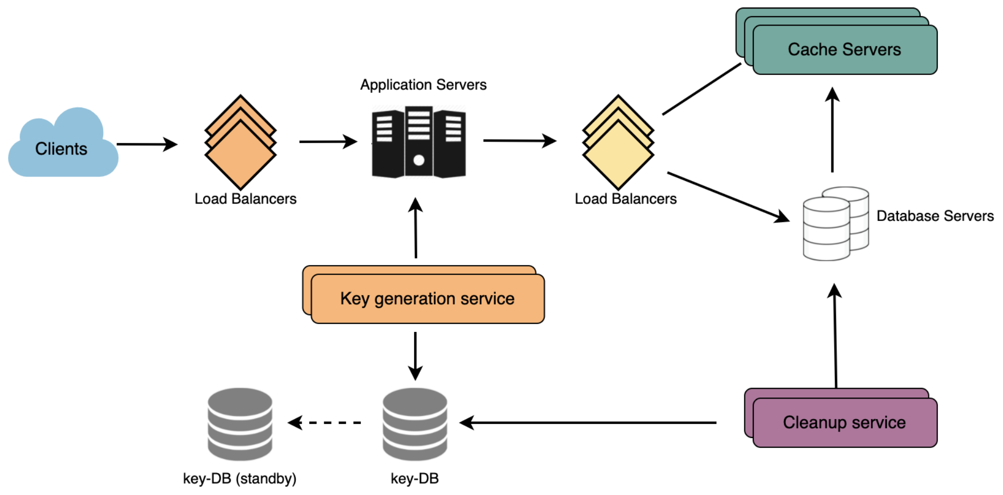
A problem with Round Robin LB is that we do not consider the server load. As a result, if a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load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

## 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 happen to the link?

If we chose to continuously search for expired links to remove them, it would put a lot of pressure on our database. Instead, we can slowly remove expired links and do a lazy cleanup. Our service will ensure that only expired links will be deleted.

- 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 storage and cache. This service should be very lightweight and scheduled to run only when the user traffic is expected to be low.
- After removing an expired link, we can put the key back in the key-DB to be reused.



Detailed component design for URL shortening