

<  educative

EXPLORE



MY COURSES



TEACH

Grokking the System Design Interview

58% COMPLETED

System Design Problems

- System Design Interviews: A step by step guide
- Designing a URL Shortening service like TinyURL**
- Designing Pastebin
- Designing Instagram
- Designing Dropbox
- Designing Facebook Messenger
- Designing Twitter
- Designing Youtube or Netflix
- Designing Typeahead Suggestion
- Designing an API Rate Limiter (*New*)
- Designing Twitter Search
- Designing a Web Crawler
- Designing Facebook's Newsfeed
- Designing Yelp or Nearby Friends
- Designing Uber backend
- Design BookMyShow (*New*)
- Further reading

Glossary of System Design Basics

- System Design Basics
- Load Balancing
- Caching
- Sharding or Data Partitioning
- Indexes
- Proxies
- Queues
- Redundancy and Replication
- SQL vs. NoSQL
- CAP Theorem
- Consistent Hashing
- Long-Polling vs WebSockets vs Server-Sent Events
- Key Characteristics of Distributed Systems
- Why System Design Interviews?

Contact Us

- [Feedback](#)

Designing a URL Shortening service like TinyURL

Let's design a URL shortening service like TinyURL. This service will provide short

Similar services: bit.ly, goo.gl, qlink.me, etc.

Difficulty Level: Easy

1. Why do we need URL shortening?

URL shortening is used to create shorter aliases for long URLs. We call these shortened a the original URL when they hit these short links. Short links save a lot of space when disp. Additionally, users are less likely to mistype shorter URLs.

For example, if we shorten this page through TinyURL:

<https://www.educative.io/collection/page/5668639101419520/5649050225344512/>

We would get:


<http://tinyurl.com/jlg8zpc>

The shortened URL is nearly one-third of the size of the actual URL.

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

If you haven't used tinyurl.com before, please try creating a new shortened URL and see the options their service offers. This will help you a lot in understanding this chapter better.

2. Requirements and Goals of the System

 *You should always clarify requirements at the beginning of the interview. Be sure to understand the system that the interviewer has in mind*

Our URL shortening system should meet the following requirements:

Functional Requirements:

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

Non-Functional Requirements:

- 1. The system should be highly available. This is required because, if our service is down, users can't access their links.
- 2. URL redirection should happen in real-time with minimal latency.
- 3. Shortened links should not be guessable (not predictable).

Extended Requirements:

- 1. Analytics; e.g., how many times a redirection happened?
- 2. Our service should also be accessible through REST APIs by other services.

3. Capacity Estimation and Constraints

Our system will be read-heavy. There will be lots of redirection requests compared to new URL shortenings. The ratio between read and write is approximately 100:1.

Traffic estimates: If we assume we will have 500M new URL shortenings per month, we need to estimate the number of redirections during that same period. What would be Queries Per Second (QPS) for our system?

New URL shortenings per second:

$$500 \text{ million} / (30 \text{ days} * 24 \text{ hours} * 3600 \text{ seconds}) \approx 20$$

URL redirections per second, considering 100:1 read/write ratio:

$$50 \text{ billion} / (30 \text{ days} * 24 \text{ hours} * 3600 \text{ sec}) \approx 19$$

Storage estimates: Let's assume we store every URL shortening request (and associated original URL) in a database. If we expect to have 500M new URLs every month, the total number of objects we expect to store is:

$$500 \text{ million} * 5 \text{ years} * 12 \text{ months} = 30 \text{ billion}$$

Let's assume that each stored object will be approximately 500 bytes (just a ballpark estimate). The total storage needed is:

$30 \text{ billion} * 500 \text{ bytes} = 15 \text{ TB}$	
URL Shortenings per month	500
Total years	5
URL object size	500
<hr/>	
Total Files	30
<hr/>	
Total Storage	15

Bandwidth estimates: For write requests, since we expect 200 new URLs every second, the total outgoing bandwidth is:

$$200 * 500 \text{ bytes} = 100 \text{ KB/s}$$

For read requests, since every second we expect ~19K URLs redirections, total outgoing bandwidth is:

$$19K * 500 \text{ bytes} \approx 9 \text{ MB/s}$$

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

Since we have 19K requests per second, we will be getting 1.7 billion requests per day:

$$19K * 3600 \text{ seconds} * 24 \text{ hours} \approx 1.7 \text{ billion}$$

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

$$0.2 * 1.7 \text{ billion} * 500 \text{ bytes} \approx 170GB$$

High level estimates: Assuming 500 million new URLs per month and 100:1 read:write ratio, the system needs to handle 19K QPS, store 15TB of data, and cache 170GB of memory.

level estimates for our service:

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

4. System APIs



Once we've finalized the requirements, it's always a good idea to define the s what is expected from the system.

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

```
creatURL(api_dev_key, original_url, custom_alias=None, user_name=None, exp
```

Parameters:

api_dev_key (string): The API developer key of a registered account. This will be used to 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, it 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

How do we detect and prevent abuse? A malicious user can put us out of business by cc design. To prevent abuse, we can limit users via their api_dev_key. Each api_dev_key can creations and redirections per some time period (which may be set to a different duration

5. Database Design



Defining the DB schema in the early stages of the interview would help to u components and later would guide towards the data pa

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
4. Our service is read-heavy.

Database Schema:

We would need two tables: one for storing information about the URL mappings, and one link.

URL		User	
PK	Hash: varchar(16)	PK	UserID: int
	OriginalURL: varchar(512)		Name: varchar
	CreationDate: datetime		Email: varchar(
	ExpirationDate: datetime		CreationDate: c
	UserID: int		LastLogin: date

What kind of database should we use? Since we anticipate storing billions of rows, and between objects – a NoSQL key-value store like Dynamo or Cassandra is a better choice. scale. Please see [SQL vs NoSQL](#) for more details.

6. Basic System Design and Algorithm

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

In the TinyURL example in Section 1, the shortened URL is “<http://tinyurl.com/jlg8zpc>”. 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 encoding could be base36 ([a-z, 0-9]) or base62 ([A-Z, a-z, 0-9]) and if we add ‘-’ and ‘.’, reasonable question would be: what should be the length of the short key? 6, 8 or 10 char

Using base64 encoding, a 6 letter long key would result in $64^6 = \sim 68.7$ billion possible s
Using base64 encoding, an 8 letter long key would result in $64^8 = \sim 281$ trillion possible

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

If we use the MD5 algorithm as our hash function, it'll produce a 128-bit hash value. Afte having more than 21 characters (since each base64 character encodes 6 bits of the hash va

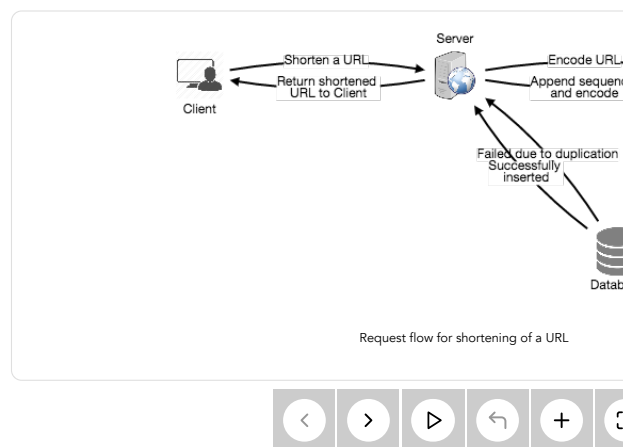
characters per short key, how will we choose our key then? We can take the first 6 (or 8) characters, but there is a risk of duplication though, upon which we can choose some other characters out of the encoding.

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

1. If multiple users enter the same URL, they can get the same shortened URL, which is not ideal.
2. What if parts of the URL are URL-encoded? e.g., <http://www.educative.io/distributed.php%3Fid%3Ddesign> and <http://www.educative.io/distributed.php%3Fid%3Ddesign> are identical except for the encoding.

Workaround for the issues: We can append an increasing sequence number to each input URL, generate a hash of it. We don't need to store this sequence number in the databases, though it could be an ever-increasing sequence number. Can it overflow? Appending an increasing sequence number could improve the performance of the service.

Another solution could be to append user id (which should be unique) to the input URL. I would have to ask the user to choose a uniqueness key. Even after this, if we have a conflict, we get a unique one.



b. Generating keys offline

We can have a standalone Key Generation Service (KGS) that generates random six letter database (let's call it key-DB). Whenever we want to shorten a URL, we will just take one key from the database. This approach will make things quite simple and fast. Not only are we not encoding the URL, but we also avoid 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. If there are multiple servers reading keys concurrently, we might get a scenario where the same key is used multiple times. 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 all the keys and one for all the used keys. As soon as KGS gives keys to one of the servers, it can move those keys to the used keys table. This way, the server can always keep some keys in memory so that it can quickly provide them whenever a server requests a key.

For simplicity, as soon as KGS loads some keys in memory, it can move them to the used keys table. If KGS dies before assigning all the loaded keys to some server, we will be left with a 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 have a 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 keys. If we store one alpha-numeric character, we can store all these keys in:

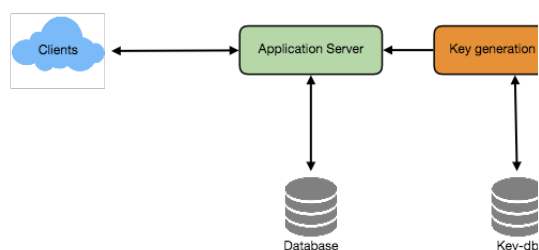
$$6 \text{ (characters per key)} * 68.7 \text{B (unique keys)} = 412 \text{B}$$

Isn't KGS the single point of failure? Yes, it is. To solve this, we can have a standby server. If 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. If a server dies before consuming all the keys, we will end up losing those keys. This could be a problem for letter keys.

How would we perform a key lookup? We can look up the key in our database or key-DB. If the key is present, issue an "HTTP 302 Redirect" status back to the browser, passing the stored URL. If the key is not present in our system, issue an "HTTP 404 Not Found" status, or redirect to the original URL.

Should we impose size limits on custom aliases? Our service supports custom aliases. Using a custom alias is not mandatory. However, it is reasonable (and often desirable) to ensure we have a consistent URL database. Let's assume users can specify a maximum of 100 characters for a custom alias (reflected in the above database schema).



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.

partitioning scheme that would divide and store our data to different DB servers.

a. Range Based Partitioning: We can store URLs in separate partitions based on the first letter. We save all the URLs starting with letter '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 accessed URLs into a single partition. We should come up with a static partitioning scheme so that we can always store URLs in the same partition.

The main problem with this approach is that it can lead to unbalanced servers. For example, we might store all URLs starting with letter 'E' into a DB partition, but later we realize that we have too many URLs that start with 'E' and need to move them to another partition.

b. Hash-Based Partitioning: In this scheme, we take a hash of the object we are storing. We can then divide the data into partitions based upon the hash. In our case, we can take the hash of the 'key' or the actual URL to determine which partition it belongs to.

Our hashing function will randomly distribute URLs into different partitions (e.g., our hash function could return a number between [1...256]), and this number would represent the partition in which we store the data object.

This approach can still lead to overloaded partitions, which can be solved by using [Consistent Hashing](#).

8. Cache

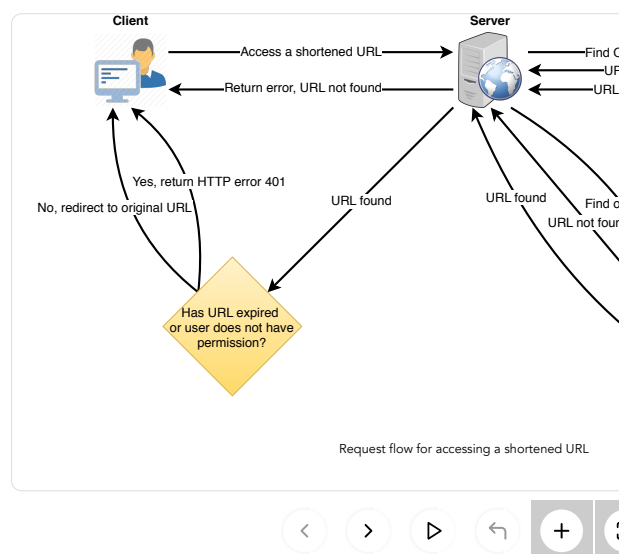
We can cache URLs that are frequently accessed. We can use some off-the-shelf solution like Redis or Memcached to store URLs with their respective hashes. The application servers, before hitting backend storage, can check the cache for the URL.

How much cache should we have? We can start with 20% of daily traffic and, based on that, estimate the number of cache servers we need. As estimated above, we need 170GB memory to cache 20% of daily traffic. If our cache servers can have 256GB memory, we can easily fit all the cache into one machine. Alternatively, we can use multiple machines to store all these hot URLs.

Which cache eviction policy would best fit our needs? When the cache is full, and we want to store a new URL, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our cache. We can evict the least recently used URL first. We can use a [Linked Hash Map](#) or a similar data structure to implement LRU. We will also keep track of which URLs are accessed recently.

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

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



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 across multiple servers in a round-robin fashion. This approach is simple to implement and does not introduce any overhead. Another benefit of this approach is that it is fair and will stop sending any traffic to a server that is not responding.

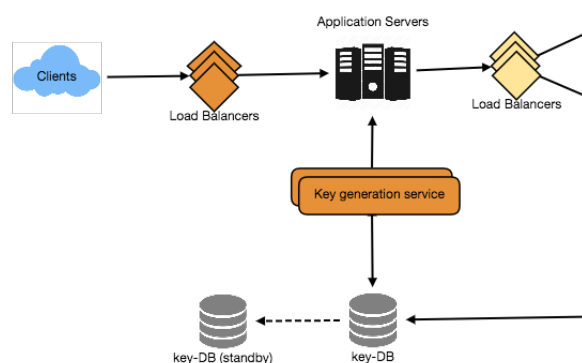
A problem with Round Robin LB is that server load is not taken into consideration. If a server is overloaded, it will continue to receive new requests. To handle this, a more intelligent LB solution can monitor the load of each server and adjust traffic based on that.

10. Purging or DB cleanup

Should entries stick around forever or should they be purged? If a user-specified expiration time is provided, the link should be purged after that time.

If we chose to actively search for expired links to remove them, it would put a lot of pressure on our service. Instead, we can slowly remove expired links and do a lazy cleanup. Our service will make sure that only a small number of expired links can live longer but will never be returned to users.

- Whenever a user tries to access an expired link, we can delete the link and return an error.
- A separate Cleanup service can run periodically to remove expired links from our service. This service can 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? If the service is getting cheap, we can decide to keep links forever.



Detailed component design for URL shortening

11. Telemetry

How many times a short URL has been used, what were user locations, etc.? How would we store this data? How often would the DB row that gets updated on each view, what will happen when a popular URL is slammed with requests?

Some statistics worth tracking: country of the visitor, date and time of access, web page that was accessed, and the IP address from where the page was accessed.

12. Security and Permissions

Can users create private URLs or allow a particular set of users to access a URL?

We can store permission level (public/private) with each URL in the database. We can also store the URL itself. If a user does not have permission and tries to access a URL, we can return a (HTTP 401) back. Given that we are storing our data in a NoSQL wide-column database, the permissions would be the 'Hash' (or the KGS generated 'key'). The columns will store the URL.

✓ Great! You've completed this section.

Not Yet

[← Previous](#)
[System Design Interviews: A st...](#)

Send Feedback or Ask a Question