Designing a URL Shortening service like TinyURL

1. Why need tinyurl?
   Users=> less mistype
   Save space to display,printed, messaged, tweeted…

2. Requirements and goals of the system

   Functional requirements: URL => unique tinyURL => redirect to the original link
   Users should optionally be able to pick a custom short link for their URL.
   Links will expire after default timespan. Users should be able to specify the expiration time.

   Non-functional requirements:
   System highly available because of ULR redirections.
   URL redirection should have minimal latency.
   Shortened links should not be guessable.

   Extended requirements:
   How many times a redirection happened?
   Service should also be accessible through rest API by other services.

3. Capacity estimation and constraints.
   Read 请求会很多，因为会 redirect，但是 write 请求也就是生成新的 URL 会比较少。 假设 100 比 1 比例。
   假设每个月有 500M 新的 URL 需要 short，那么 500M * 100 就是 50B 的 redirection 需要执行。

   QPS?
   500million/（30days * 24hours * 3600 seconds） = 200 URL/s 写请求
   读请求就是 200 * 100 = 20k/s

   存储空间估算:

假设存 5 年，每个月 500M 新的 URL， 总共 30B
假设每个存储的 object 大小是 500bytes，一共 15TB。

带宽估计：
写请求 200 个/s，所以 200*500=100KB/s.
读请求， 10MB/S

内存估计：
如果需要缓存，假设 80-20 规则，那么
每天 cache 170GB。

High-level estimates:

New URLs	200/s
URL redirections	20K/s
Incoming data	100KB/s
Outgoing data	10MB/s
Storage for 5 years	15TB

Memory for cache	170GB

4. System APIS?
   String createURL(api_key, original_url, custom_alias=None, user_name = none, expire_date = None)
   deleteURL(api_key, url_key)

   prevent abuse, limit users via their api_dev_key!!!

5. DB design
   存储大量记录，记录之间没有关联，每一条记录很小，read-heavy。
   建议使用 NOSQL，因为 read-heavy，而且不需要 object 之间的关系。
   需要两个表，URL 和 user

URL	
PK	**Hash: varchar(16)**
	OriginalURL: varchar(512)
	CreationDate: datetime
	ExpirationDate: datatime
	UserID: int

User	
PK	**UserID: int**
	Name: varchar(20)
	Email: varchar(32)
	CreationDate: datetime
	LastLogin: datatime

6. Basic system design and algorithm

a. encoding actual url

假设 base64 的六位数足够编码。

问题是：多个用户输入相同 URL 得到相同 tinyURL,

或者 URL 已经有部分编码，也会出现问题

可以 hash(url + increasing sequence) 但是会 overflow

可以 append user_id,但是用户没登录，就要让他一直选择 unique key，直到不重复。

b. generate keys offline

使用 key generation service. 不需要担心 duplicate

KGS 给了 key 之后，马上就把它移到 used key 表里。

同时，为了保证不把同一个 key 给多个 server，必须 synchronized hold key

KGS 要是 single point failure 呢？

Solution： replica of KGS。

each app server cache some keys from key-DB，是可以的，因为数量非常多，不怕损失 key。

How would we perform a key lookup?

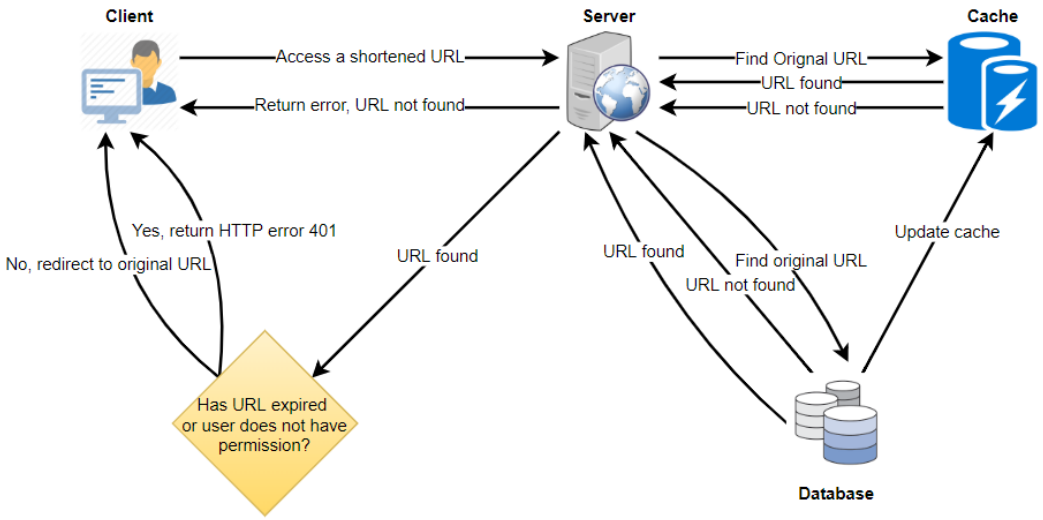
在数据库里查，查到就返回 302redirect，查不到就 404

Should we impose size limits on custom aliases? Yes, 16 characters limit is reasonable.

7. **Data partitioning and replication**
   Range-based partitioning / hash-based partitioning
   Range-based partition=> LB problem
   Hash-based => still overhead=> use consistent hashing

8. **Cache**

   Size : about 170 GB,
   Policy : LRU used by using linkedhashmap (linkedlist + hashmap)



9. **Load balancer**

   Between Clients and Application servers

   Between Application Servers and database servers

   Between Application Servers and Cache servers

   使用 round-robin + periodically query the server's load and adjust traffic on it.

10. DB clean up

    Entry expire =》 clean up

    轻量级 人少的时候运行的 clean up

11. Telemetry

    Statistics info

    Some statistics worth tracking: country of the visitor, date and time of access, web page that refers the click, browser, or platform from where the page was accessed.

12. Security and permissions

    store permission level (public/private) with each URL in the database

    store permission level (public/private) with each URL in the database

    key : hash(url)

    value: List<userId> which has permission of this url