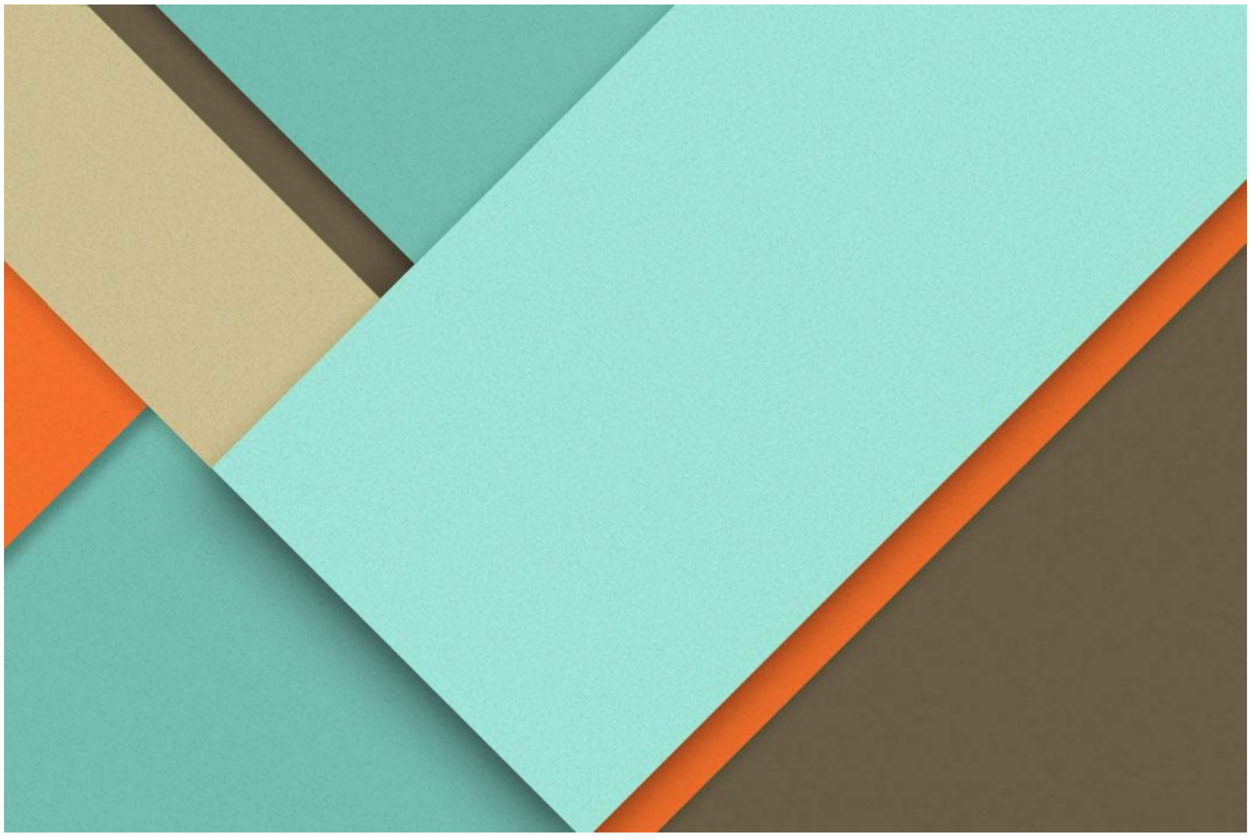


URL Shortener

26.04.2021

Sanni Heruwala

Overview

Url Shortener is a service that creates a short alias URL for a corresponding long URL. So, now whenever the user visits the short URL, he will be redirected to the original URL.

Goals

The goal of the project is to design a Url shortener service like bit.ly that is real time scalable.

Specifications

1. Customer has to login first time and a token will be given to each.
2. 1,000 daily active users creating 5 to 10 short URLs per day.
3. Predicted daily volume of 100,000 daily visitors clicking the short URL.
4. Use link.ks as domain
5. Shortened URLs must be unique for each customer.
6. Two customers can have the same long URL.
7. Duplicate shortened links for each customer are not allowed. If a customer attempts to create a new shortened link for a URL that already exists, the existing shortened link will be provided.
8. Report usage for each shortened link
    a. Total number of clicks(redirect)
    b. Number of clicks by day
    c. Total number of people who clicked links
    d. Total number of people who clicked by day

Data capacity model

Estimated clicks and url creation:-

Items	Per day	Per month	Per year	Per 5 year
Daily active users	1000	30,000	360,000	1,800,000
URL creation	DAUx10=10,000	300,000	3,600,000	18,000,000
URL request	100,000	3,000,000	36,000,000	

User data :-

Column	Size
Token/User id	7 bytes
Name	100 bytes
Email	100 bytes
Password	14 bytes

User data could grow 221 bytes * 30000 = 6.6 MB per month and nearly 80 MB per year.

Short link related data:-

Column	Chars	Size	Comments
Token/User id	7	7 bytes	
Long url	2048	2 KB	
Short url	19	19 bytes	www.link.ks/abcPQR1
Creation time	10	10 bytes	epoch
Expiry time	10	10 bytes	epoch

Short link related data could grow 2.094KB * 300,000 = 628 MB per month and nearly 7.5 GB per year basis.

Data Storage Observation

1. We will be storing 300K of links per month.
2. The short link redirection should be fast.
3. Our service is going to be read heavily(3 millions per month).
4. The only relation that is going to exist is which user created which URL and that too is going to be accessed very less.

We have two different choices of databases:

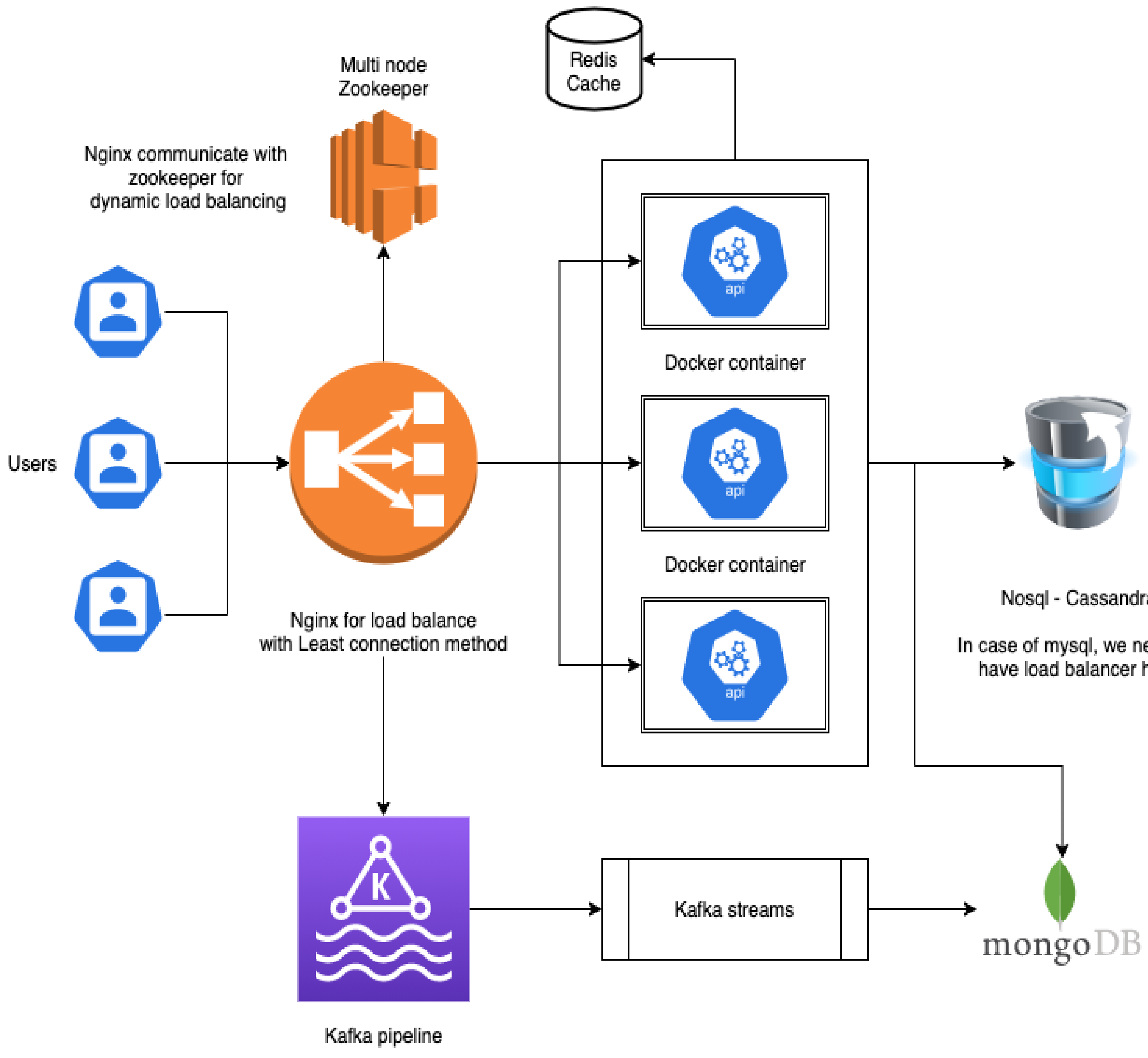
1. Relational Databases(MySQL)
2. NoSQL Databases.

In general, Relational Databases are good if we have lots of complex queries involving joins, but they are slow.

NoSQL databases are pathetic at handling the relationship queries but they are faster.

Now, we don’t really need lots of relationships among data, but we do need the fast read and write speed. Hence we will **choose NoSQL Database**. The key for each row can be the shorturl, because it is going to be globally unique.

System Architecture


Redis Cache
Multi node Zookeeper
Nginx communicate with zookeeper for dynamic load balancing
Users
Docker container
Docker container
api
api
api
Nosql - Cassandra
In case of mysql, we need to have load balancer here
Nginx for load balance with Least connection method
Kafka streams
mongoDB
Kafka pipeline

Technique Create short URL - Base 62 encode with mysql

Base 62 provides nearly 3.5 trillion combinations, which result in rare collisions for our case.

1. Receive create url request
2. Load balancer redirect the request to relevant api server
3. Check long url exist in db for same user(token)
4. If no, Generate random number
5. Apply base 62 encode and generate 7 digit alphanumeric value
6. Append with domain name
7. Before insert into db, run mysqlDB level check insert if absent short url
8. If short url exists then repeat from step 4

Technique Create short URL - Use counter range from Zookeeper with noSQL DB and Redis

In order to scale our application, a distributed system fits better in the scenario.

Here we will be using a multi node zookeeper to manage the range of counters for each api server. It guarantees no collision.

Only drawback is , cassandra is eventually consistent but that can be resolved with proper consistency and replication factor.

We will use Redis as a cache with LRU policy.

1. As soon as a new server gets added, zookeeper will assign a range of counters to that server, which it will use to generate a number.
2. Meanwhile nginx will also communicate with zookeeper to add a new server in its list.
3. Nginx will be using RR or Least connection methods for load balancing between the servers.

4. For creating an url request, API will check Long url in redis which is a distributed cache for this system.
5. Let say url exists for the user/token, it will return the same old short url.
6. If it does not exist then it will generate a counter number and encode it with base 62 and create a short url.
7. This short url will be then inserted into cassandra/nosql with a respected user token where the short url will be key clustered by user token and rest other information.
8. As soon as the counter range is consumed by the server it will request a new range to the zookeeper.

Get long URL

Since Redis is maintaining the LRU policy and keeping trendy URL in its key value store, it will be a quick lookup into it.

1. Get request with corresponding short url
2. Pass it through load balancer
3. API will check in redis for short url
4. If not exist it will check in cassandra or mysql
5. Redirect to the URL

Update long URL

It will take a user token, short url, old long url and new long url as an input request.

1. User request will be forwarded via load balancer to api
2. API will read from cassandra and search based on short url
3. Short url will update the data based on the old long url

Delete URL

It will take a short url and token as an input.

1. User request will be forwarded via load balancer to api
2. API will delete relevant entry for a short url with matching token.

Analytics with each Get Long URL request

Request timestamp - epoch - 10 bytes

Short url - 19 bytes

User_details - generate hash code from all the information - 7 bytes

Token - 7 bytes

The data would grow 130 MB per month.

- For each get long url request, nginx will send that event encapsulated with request timestamp, short url and requester's information to kafka
- From kafka that would be processed with kafka stream and it would retrieve user token from cassandra/user table.
- Then the final json document will stored into mongodb

Generate report

It will take a **user token** as an input and return aggregated values for each shortened URL.

Tech details

- Scala/Python for API using either flask or akka
- Docker container
- K8 for docker management and scale
- Nginx as load balancer
- Zookeeper for dynamic load balance and range counter manage
- Redis for cache and quick read
- Cassandra for data store of user and short link creation
- Mysql db for replacement of cassandra in case of low usage
- Kafka for analytics events
- mongoDB for event storage

Estimation and timeline

With the team of 3 dev - 1 devops and 1 tester

Infra setup and api development - 15 story points

Integration testing - 4 story points

Stag testing - 4 story points

Final deployment and documentation - 3 story points