Designing a URL Shortening service

Introduction:

MUS (Mobileye URL Shortener) is a URL shortening web service that is responsible for creating shorter aliases for long URLs.

System Requirements:

Functional Requirements:

Our service should generate a shorter and unique from a long URL.
Redirect in real time the users when they want to access the original link by the short URL.
Users can enter a custom short link for their URL.
Accessible through REST APIs.

Non-Functional Requirements:

Perfect Error handling the server should return only HTTP codes in the range of > 200 < 499.
If the user logged in see the list of his shortener URLs.

System Capacity and constrains:

Traffic:

Assuming we will have 100M new unique URL shortenings per month

100M / (30 d * 24 h * 3600 s) = 40 unique short URLs per second

Assuming a 100:1 read/write ratio

number of redirections requests = 40 URLs/s * 100 = 4000 URLs/s

Storage:

Let's assume:
50 years = The lifetime of service
100M shortened links created per month
500 bytes = The size of each object (long URL, short ID, Date, user ID)

The number of data objects in the system will be:

100 million/month * 100 (years) * 12 (months) = 120 Bllion

The total storage = 120 b * 500 bytes = 60 TB

Memory:

Following the 80:20 rule for caching. Where 80% of requests are for 20% of data.

Assuming we get 5000 requests per second, then the number of requests per day will be:

5000 * 3600 seconds * 24 hours = 450M

To cache 20% of requests:

20% * 45M * 500 bytes = 45 GB

REST APIs

Let's starts by making two functions accessible through REST API:
create(longURL, shortId , userId , creationDate)

POST

Request Body: {url=long_url , short_id= shortID }
Return the short Url generated, or error code in case of the inappropriate parameter.

long_url: A long URL that needs to be shortened.
shortId: (optional): The custom id that the user wants to use.
userId: if the user is login, then send the user ID
creationDate: date of creating the object

GET: /{short_id}

Return the short Url generated, or error code in case of the inappropriate parameter.
short_id: The short ID generated from the above function.

Database:

We need to store millions of records and each object is less than 1k byte. If we consider there is no relationship between records. We need to store the user who created the URL and share its id with the created link we don't need to use join. This leads us to build NoSQL Database with the following schema:

URL Table:

Short id (pk)	Varchar(16)
Long URL	Varchar()
Creation date	datetime
User id	Int

User Table:

User id (pk)	Int
User name	Varchar()
Creation date	datetime
Email	Varchar()

We are expecting to maintain millions of URLs monthly - need to maintain big storage - we can leverage MongoDB for scaling across a different instance. And for DOM we can use the Mongo engine to build the classes.

Shortening Algorithm:

We can generate a shortened URL from the provided URL by computing MD5 hashing that produces a 128-bit hash value. The length of a short ID using the MD5 hashing is 7 bytes (letters). But if we encode the hashing value by using Base64 we'll get a string having a number of 68.7 billion possible strings with long more than 21 characters in the string. we can take the first 6 letters of the key.

To resolve the duplication or collisions, we can add the current date for the string to make it unique and then generate its hash

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

MD5(long URL) -> base62encode(128-bit hash value)

Cache

We need to speed up our database reads by cache URLs that are accessed. Our cache could store, let's say, 20% of the most used URLs. We can use some off-the-shelf solutions like Memcached, which can store full URLs with their respective hashes. When the cache is full, we would want to replace a URL with a more popular one. So to solve that Least Recently Used (LRU) cache eviction system would be a good choice and can be a reasonable policy for our system. A linked Hash Map or a similar data structure can be used to store our URLs and Hashes, which will also keep track of the URLs that have been accessed recently.