Project Report

NanoURL *- A URL Shortener*

Harsh Vasoya

24th January, 2019

Table of Contents

Overview

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. Url shortener service is used to share shortened url via SMS or link via mail. URL shortening is used for optimizing links across devices, tracking individual links to analyze audience and campaign performance, and hiding affiliated original URLs.

In this project, I have created *NanoURL* - a service which will primarily shorten url for a given long url and give long url for a short url.

Requirements

This URL shortening system ensembles following requirements:

1. Given a URL, this service will generate a unique, shorter alias of it, called as short URL.
2. When users click on the short URL, this service will return corresponding long URL if it is existing or will give an error message, otherwise.
3. Each short URL will get expire after a predefined amount of time and will be removed from the storage.
4. Analytic report will get generated on the day-to-day basis.

Technical Specifications

NanoURL service utilises following technologies for the development and proper functioning:

1. Language - Java
2. Framework - Spring
3. Database - MySQL
4. Server - Apache Tomcat
5. Cache - Redis

Unique Key Generation and Encoding

The process of conversion of long URL to short URL has two important stages - Unique Key Generation and Encoding. Unique Key Generator (UKG) provides a unique finite number which is passed down to Encoder for encoding it into valid character set.

Unique Key Generation

There were several UKGs which were taken into the consideration which are as follows:

1. Random number key
2. Incremental number key
3. Time-based key
4. Hash-based key (like MD5 or SHA256)
5. Key Generation Service (KGS)

Out of these many options, time-based key generation technique was selected because here number was generated based on the difference in milliseconds of the current time with the midnight of January 1, 1970 (UTC) - which ensures the uniqueness of the key at any point of time which is guaranteed in Random number keys. Moreover, unlike KGS and Incremental keys, server doesn't have to store any values during the execution and thereby making it "stateless". This technique also provides the flexibility of having different short URLs for the same long URLs which can be used for multi-campaign performance analysis. This feature is unavailable in Hash-based Key Generators.

Encoding

Encoder converts the large number provided by the UKG to a small alphanumeric string. This conversion takes place by repeatedly doing the modulo operation of large number with the base value and converting the remainder into its corresponding character.

In this project, I have used base62 ([A-Z, a-z, 0-9]) encoding to make short URL simple and easy to read. Using these characters, system will generate 7-length string. Reason behind selecting 7-length string for short URL is because it would provide 62^7 (= 3.5 x 10^{12}) unique combinations, which would be sufficient enough.

One more aspect that was taken into consideration was that after a point of time, UKG will start generating a number which will be greater than the maximum number that a 7-length base62 encoded string could hold (62^7). In order to avoid this issue, time number is wrapped around a largest prime lesser than 62^7 *(= 3,521,614,606,199)*. Prime number was chosen over composite number as it would provide non-repetitive mapping and will almost cover all the possible values.

Database Design

MySQL is used as the database for storage purpose. There are two tables in the database schema: one for storing information about URL mappings and one for storing the daily generated reports.

Table Schema

Urls	
id	*bigint(20)*
short_url	varchar(22)
long_url	varchar(2084)
created_at	datetime
updated_at	datetime

Reports	
id	*bigint(20)*
created_at	datetime
num_total_clicks	bigint(20)
num_unique_clicks	bigint(20)
num_short_url_created	bigint(20)
num_long_url_added	bigint(20)

Attribute description:

Urls

- **id** - unique number for each record of the table (Primary Key, Auto-Increment)
- **short_url** - unique short link for the corresponding long URLs
- **long_url** - the original URL
- **created_at** - timestamp of record creation
- **updated_at** - timestamp of most recent record updation. This field is updated when user clicks on short URL.

Reports

- **id** - unique number for each record of the table (Primary Key, Auto-Increment)
- **created_at** - timestamp of record creation
- **num_total_clicks** - count of total number of clicks on all short URLs
- **num_unique_clicks** - count of total number of unique clicks on all short URLs
- **num_short_url_created** - count of total number of unique short URLs created
- **num_long_url_added** - count of total number of distinct long URLs added

Note: All the counts in the above schema are for a single day.

Indexes

As this service is *read-heavy*, indexes were created on several columns of both tables for rapid random lookups and efficient ordering of access to records.

Table	**Columns**
Urls	id short_url updated_at
Reports	id created_at

System APIs

System will follow REST API nomenclature for defining APIs. Following table shows the list of APIs which are defined in the NanoURL system.

POST - /NanoURL/url	
Input	Request-Body JSON { "longUrl": String }
Output	JSON { "shortUrl": String, "longUrl": String }
Service	createUrl (newUrl)

GET - /NanoURL/url	
Input	Request-Param("shorturl" - String)
Output	String (long url)
Service	getLongUrl (shortUrl)

DELETE - /NanoURL/url	
Input	NA
Output	NA
Service	removeOldUrls ()

GET - /NanoURL/report	
Input	Request-Param ("date" - String) [format: dd-mm-yyyy]
Output	JSON { "numTotalClicks": Number, "numUniqueClicks": Number, "numShortUrlCreated": Number, "numLongUrlAdded": Number}
Service	getReportByDate ()

POST - /NanoURL/report	
Input	NA
Output	JSON { “numTotalClicks”: Number, “numUniqueClicks”: Number, “numShortUrlCreated”: Number, “numLongUrlAdded”: Number}
Service	generateReport ()

Additional Functionalities

Caching

In order to improve the response time for GET requests of long URLs for given short URLs, cache memory is also integrated into the system. Redis server has been used as the cache server. It stores key-value pairs of the short URL(input) and long URL(output) in its memory for a specific timespan to serve the caching purpose.

Cron Services

Various cron-jobs were added to the system for automating various tasks. This tasks are scheduled at a specific interval of time to execute certain jobs. Detailed list of cron services is as follows:

Service	**Scheduled Time (hh:mm:ss)**	**Task Description**
addTodayRecord()	00:00:00	Adds a new record in Report table in order to count clicks
removeExpiredUrls()	00:00:10	Removes expired URLs from Url table
generateReport()	23:55:00	Generates the report for the current day

Note: All these jobs are executed daily at the specified time.

Limitations and Scopes of Improvement

Following are some limitations that might hamper the proper and efficient functioning:

1. This system can handle requests of creating short URL at maximum rate of 1 request for each millisecond. If more than one requests are made on the server for the same millisecond, then same short URLs will be created for different long URLs.
2. This system requires single web server for functioning and hence, it will not work on distributed server environments.
3. Caching technique used is very static. More improvement can be done on cache eviction strategies such as implementing Least Recently Used (LRU) policy.
4. Generated reports are inaccessible apart from the API access. Mail Delivery System can be integrated for sending daily reports to respective authority via mail.
5. There is no authentication layer for the system and thereby, it is vulnerable to cyber attacks.