URLShortener System Design Document

System Design Document

Prepared By: Aryan Tamang

Version: 1.0

Date: 1st July, 2026

Status: Draft

Table of Contents

1. Requirements....3
   1.1 Functional Requirements....3
   1.2 Non Functional Requirements....3
2. Scale Estimation....4
3. Architecture....5
   3.1 Actors....5
   3.2 Components and why they exists?....5
   3.3 Data Flow Diagram....6
4. Schemas....7
5. BottleNecks....8
6. Improvements....9

1. Requirements

The main purpose of this URLShortener application is to shrink the size of links so that more of the content fits in in limited place.

1.1 Functional Requirements

The features or functionalities that this application must illustrate are:

1. Create Short URL.
2. Redirect Short URL to Long URL
3. Count the Clicks to the URL, Along with other analytics.
4. Expire the URL.
5. Social Seller Page (Esewa, Whatsapp,Instagram)
6. Branding and Dynamic QR(Very helful for hospitality industries)
7. Monitization (Subscription, limit, payment integration, Esewa and Khalti for now)

1.2 Non Functional Requirements

These are not the direct functionalities of the application but helps the application in indirect manner. The following points justifies the non functional requirements of this application:

1. High Availability
2. Low Latency (<100ms)
3. High Security
4. Confidentiality
5. Proper Access Control Mechanism

Engineer Note: *The first step in an engineering mindset is to gather the requirements properly so that we thoroughly understand the problem and prevent us from building the wrong thing.*

2. Scale Estimation

I consider this phase as one of the most important part while designing any system. The process of trying to estimate the approximate users or resources intended for the application is called scale estimation. The scale estimations for this application are:

1. 110M urls stored.
2. 50k Users per day.

If we are estimating around 100M records. And Let's assume single record occupies are 1000 bytes.

Then total storage = 1000 bytes * 110 M = 110GB

Therefore, Single Database would be fine for it.

URLShortener is read-heavy application. Let's assume a 10:1 ratio where an average user creates 3 links but redirect happens 10 times as often.

Write QPS = 50k Users * 3 links = 150k Links Creations per day = 150k / 86400 = 1.734 requests / second

Read QPS = 150k * 10 = 1.5M redirects per day = 1.5M / 86400 = 17.36 requests / second.

Peak QPS = 17.36 * 10 = 170 requests / second.

For a simple primary index reads, Postgresql can handle upto 10,000 – 50,000 requests/second.

Engineering Note: *The main motto of scale estimation is to prevent us from overengineering.*

3. Architecture

Architecture refers to the high level design of our application. It defines what components exists in our application and how they work together as a system.

3.1 Actors

Actors refer to the entity who are going to be engaged within the application directly or indirectly. They are:

- Users.

- Browsers.

- Analytics Reporter.

3.2 Components and why they exists?

Componenets refers to the building blocks that makes up the entire application. They are:

I. Client

II. Browser

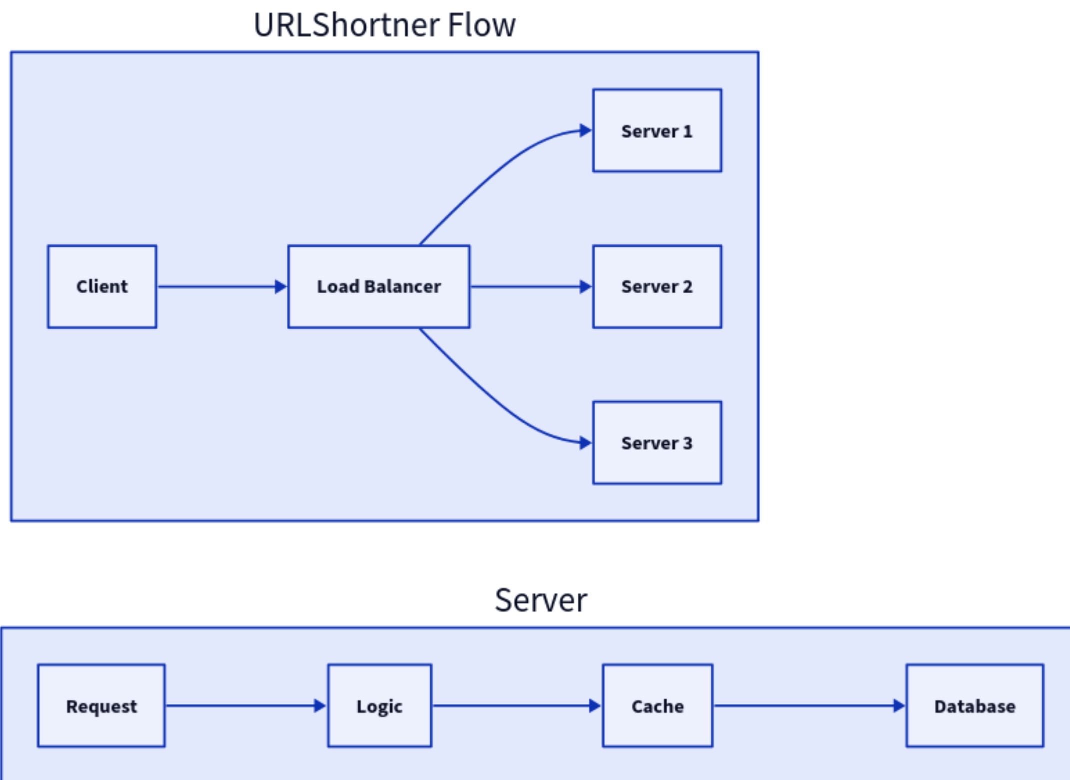
III. Load Balancer

IV. Cache

Since our requirement is to reduce latency as much as possible. Adding Load Balancer helps us in both maintaining availability and reducing latency.

We estimate around 50k users to use our application daily. So heavy database will slow our application. Thus we move as much as possible of work from direct database by implementing caching. Also we can implement read replicas and one primary write in the coming future.

3.3 Data Flow Diagram



4. Schemas

```
Table sys_urls {
  id integer [primary key]
  long_url text
  short_code varchar(150)

  created_at datetime
  expires_at datetime
  click_counts integer
}
```

Engineering Note: *I've seen a lot of mistakes regarding data modeling. Modeling data in the later stage tends to lead to lots of bugs in the software development life cycle. So it is wiser to model the database beforehand.*

5. BottleNecks

While developing this application, certain challenges might occur. The following points are some of the assumptions of the problems that may arise during the lifecycle of this application:

I. URL Uniqueness

II. High Latency

III. Stale Cache Data

Engineering Note: *This step helps you to anticipate what can fail, why can it fail?*

6. Improvements

We can implement following improvements to try to mitigate above bottlenecks:

I. For URL uniqueness (use time based and other random varible to maintain almost 100% uniqueness in short url)

II. To remove high latency (we use caching very intelligently)

III. Stale Cache data must be handled with proper cache invalidation techniques such as deleting on update etc.