Design TinyURL System

Practice on: 17 Jan 2025 - Friday

Time Taken: 80 Mins (1 Hour 20 Mins)

Qs for the interviewer

1. What can be length of the tinyURL?
2. System is for a B2B or B2C?
3. DO we need to rate limit a user - how many tinyURL an user can create in a min?
4. Can user provide custom alias in the tinyURL?
5. Can user provide custom expiration for requested tinyURL?
6. How long system/database need to persist the tinyURL mapping?
7. Total users? DAU?
8. What will tinyURL consist of? sample? Can we have special characters?

Requirements

Fuctional Requirements

1. User provides a long URL and get a short URL.
2. User click on the shortURL and get redirected to the longURL
3. User can provide custom alias and expiration for the stored URL (OPTIONAL)

Non Functional Requirements

1. System should have low latency (return short URL under 200 ms)
2. System, should be highly available
3. System should be durable - stored URL should not be lost

Capacity Estimation

Assumptions:
Total Users: 100 million
DAU: 10 million
Read:Write ratio: 10:1
LongURL avg length: 100 characters
ShortURL avg length: 10 characters

Throughput - RPS & WPS
WPS = 10 million * 1/10 = 10^6 => 10^6/24*60*60 ~ 12WPS
RPS = 12WPS * 10 = 120 RPS

Storage:
stortURL = 10 bytes longURL = 100 bytes created_date = 10 bytes
Aprox ~ 150 bytes for every entry
Storage for a day: 10 million * 1/10 = 1 million writes * 150 bytes = 15 * 10^9 => 15 GB
Storage for 10 years = 15 GB * 365 * 10

Core System Entity

1. User
2. LongURL
3.ShortURL
4. Alias
5. Expiration

API

1. **User submits a longURL -> gets a shortURL in response**

REQUEST:

HTTP POST API /short
body{
long_url: "www.facebook.com/profile/angelPriya",
alias: "",
expiration: ""
}

RESPONSE:

{
short_url: "www.tiny.com/qb2323j"
}

2. User clicked on tinyURL -> gets redirect to longURL

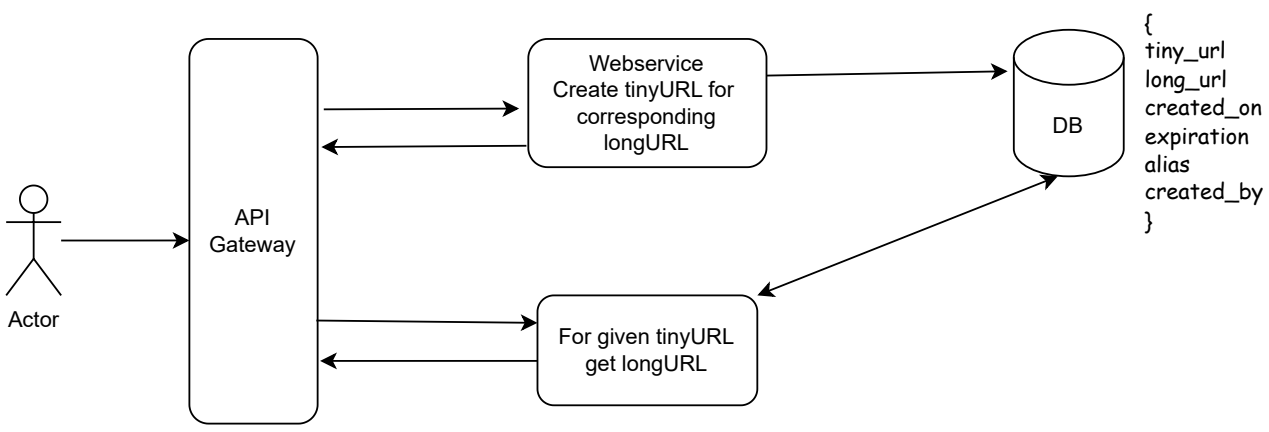
REQUEST:

HTTP GET API /qb2323j

RESPONSE:

Status code 302
{
redirect_url: "www.facebook.com/profile/angelPriya"
}

High-level-system-designe



Deep-dive

[1] Creating tinyURL for corresponding longURL

1. One way to create tiny URL can be using SHA-256 on longURL to get a hash. This hash will be long so can use the first 7 characters as the key of tinyURL
Drawback :
> their can be collisions: as first 7 char hash can be same - counter can be used but we before saving in DP we need to check the uniqness of the key each and every time.

2. We pre-calculate a lot of hash key and keep them stored in another database, whenever a new longURL request

comes we get one unused key and map that with the long user and store in the mapping DB
Pro:
> NO need to calculate/find key on the fly > this will reduce the latency
> No need to check the uniquness of the key

In unique_tinyUrl_key table we can have a boolean value that can be set false whenever its been used
Or we can have 2 DB > one will keep unused keys and other will store the used key > this is help in increasing the latency.

[2] How long can the tinyURL hash Key

12WPS > 12 * 60 * 60 * 24 ~ 1 Million tiny URL created every day
[a-z] + [A-z] + [0-9] > 26 + 26 + 10 > 62 characters
6^62 ~ ????
7^62 ~ ????

This way we can decide we the tinyURL's min length

[3] Low Latency

> Introduce cache before DB
> Once a tinyURL created probability of its been used in a short time is high, so we keep in cache
> Once a tinyURL been read from DB, put that in cache
> LRU will be the eviction policy
> caching strategy ?????
> DB stores to cache?? OR web service stores to cache??

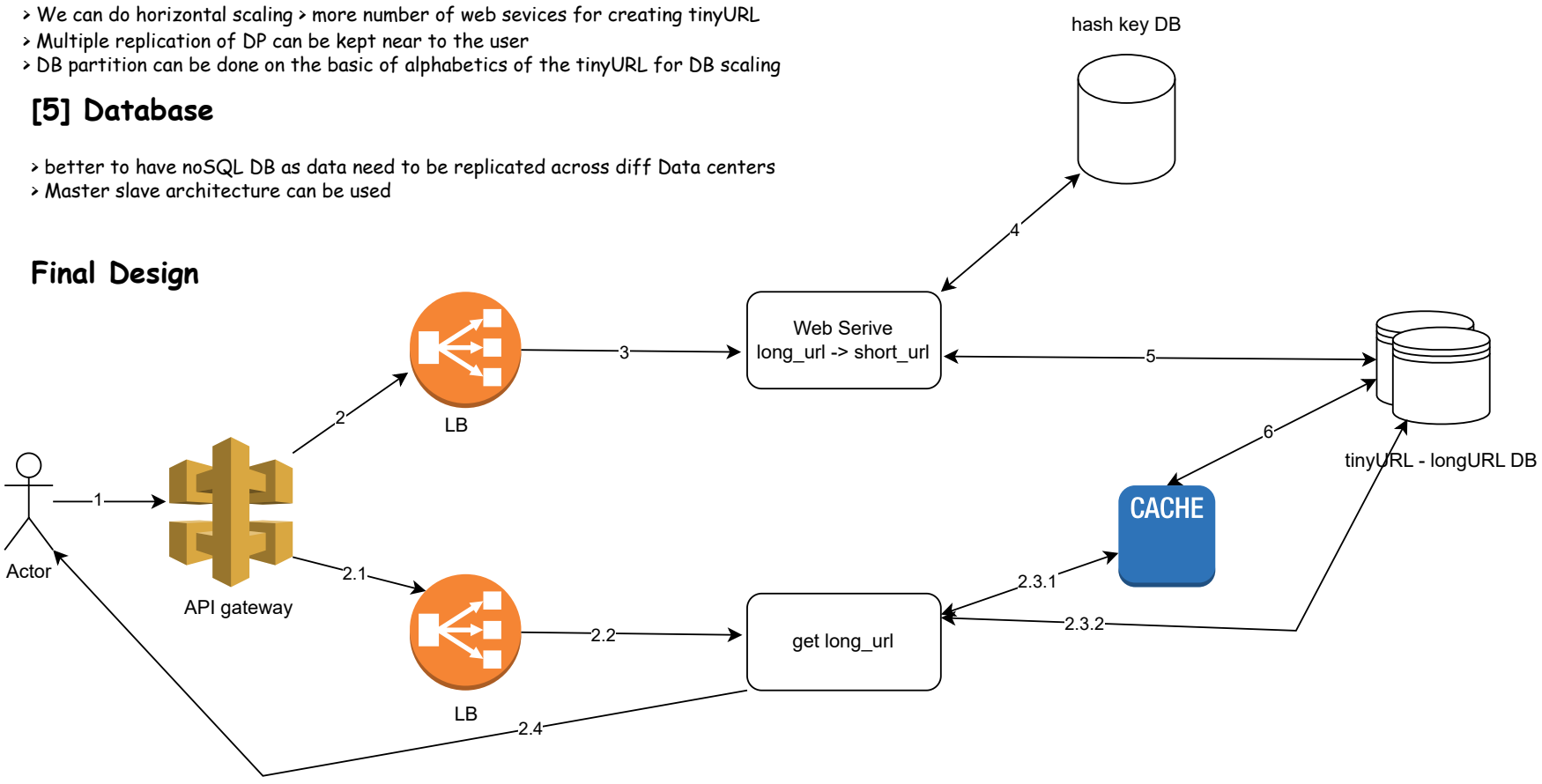
[4] Scaling

> We can do horizontal scaling > more number of web sevices for creating tinyURL
> Multiple replication of DP can be kept near to the user
> DB partition can be done on the basic of alphabetics of the tinyURL for DB scaling

[5] Database

> better to have noSQL DB as data need to be replicated across diff Data centers
> Master slave architecture can be used

Final Design



Design Review by chatGPT

Strengths:

1. **Thorough Questioning**:
   - The questions you outlined for the interviewer are well thought out and address key aspects like URL length, B2B vs. B2C considerations, rate-limiting, custom aliases, and expiration.
2. **Clear Requirements**:
   - Both functional and non-functional requirements are explicitly stated, ensuring a comprehensive understanding of the system's goals.
3. **Capacity Estimation**:
   - The calculations for throughput, reads, writes, and storage requirements are clear and logically sound.
4. **Core Entities**:
   - Identification of the core system entities (User, LongURL, ShortURL, Alias, Expiration) aligns well with the functional needs.
5. **API Definitions**:
   - The API endpoints for creating and resolving URLs are well-defined, including request and response formats.
6. **Scalability and Performance**:
   - Proposals for horizontal scaling, caching, and database replication demonstrate a strong understanding of scalability challenges.
7. **Database Design**:
   - The choice of NoSQL for replication across data centers and the use of master-slave architecture are appropriate for a system like TinyURL.

Areas for Improvement:

1. **Hash Collisions**:
   - While you mention using SHA-256 and truncating it to 7 characters, this approach requires collision handling.
     Including a more detailed collision resolution strategy (e.g., appending unique salts or retrying with different prefixes) would strengthen the design.
2. **Pre-computed Keys**:
   - The idea of pre-computed keys is efficient but introduces complexity in managing the unused keys database.
     Discuss potential bottlenecks (e.g., key exhaustion, synchronization issues) and mitigation strategies.

3. **TinyURL Length Calculation**:
    - The computation for deciding the tiny URL's length is partially done.
      Providing a complete mathematical derivation with examples would make it more robust.

4. **Cache Strategy**:
    - While caching is mentioned, the cache invalidation strategy is unclear. For instance:
        - Should expired keys be immediately purged?
        - How does cache syncing occur with the database in case of updates?
    - Clarifying the read-through or write-through caching model would improve this section.

5. **Rate Limiting**:
    - You mentioned rate-limiting as a question but didn't include it in the final design.
      Consider integrating a rate-limiting mechanism (e.g., token bucket or leaky bucket) to prevent abuse.

6. **Data Durability and Consistency**:
    - For a system with high availability requirements, discuss the trade-offs between consistency and availability in your NoSQL setup, especially in scenarios with network partitions.

7. **Security**:
    - Address potential security concerns, such as:
        - Preventing malicious URLs (e.g., phishing links).
        - Ensuring the system is protected against DDoS attacks.
        - Validating custom aliases to prevent misuse.

8. **Additional Features**:
    - Consider adding more details about optional features like analytics (e.g., tracking the number of clicks) and admin capabilities (e.g., purging old or malicious URLs).