**URL Shortening Service (e.g., TinyURL)**

1. Solidify the requirements – both Functional and Non-Functional

* Functional Requirements:
  + **Short URL generation**: Our service should be able to generate a unique shorter alias of the given URL.
  + **Redirection**: Given a short link, our system should be able to redirect the user to the original URL.
  + **Custom short links**: Users should be able to generate custom short links for their URLs using our system.
  + **Deletion**: Users should be able to delete a short link generated by our system, given the rights.
  + **Update**: Users should be able to update the long URL associated with the short link, given the proper rights.
  + **Expiry time**: There must be a default expiration time for the short links, but users should be able to set the expiration time based on their requirements.
* Non-Functional Requirements:
  + **Highly Availability**: By Incorporating Fault tolerance (failover mechanisms for load balancers) and Repetition of System Bottlenecks.
  + **Minimum Latency**: Low latency to provide the user with a smooth experience. Redundancy of servers (application and storage)
  + **Scalability**: By Incorporating Horizontal Scaling.
  + **Readability**: The short links generated by our system should be easily readable, distinguishable, and typeable.
  + **Unpredictability**: From a security standpoint, the short links generated by our system should be highly unpredictable. This ensures that the next-in-line short URL is not serially produced, eliminating the possibility of someone guessing all the short URLs that our system has ever produced or will produce. Will achieve this by Incorporating Encoding.

1. Scope the Problem

* What kind of clients? Mobile Apps, Web Browsers, Smart TVs?

1. Capacity/ Resource Estimation
2. Traffic Estimates
3. DAU – 100M
4. Writes or URLShortenings/ Month – 200M
5. Read/ Write Ratio – 100:1
6. Daily User handling limit of a server (assume that a typical Instagram server handles 100 requests per second) – 8640000
   * URLShortenings/ Sec –
   * URLRedirections/ Sec –
   * Total Servers Required – 100M/8000 =
7. Storage Estimates
8. Time Duration for which objects are required to be stored – 5 years
9. Usage/ day. URLShortenings/Month – 200M
10. File Size Requirements – Each URL Shortening requires 500 bytes

* Total Requests for 5 years =
* Total Storage Required =

1. Bandwidth Estimates

* Incoming Data –
* Outgoing Data –

1. Memory (Cache) Estimates
2. 80/20 Rule? Yes

* Cache Size = ((200M x 500B) / 30) x 0.20 =

1. System API’s

* 3 APIs needed to expose the functionality of our service

1. Database Design
2. Present the building blocks of the Design
3. Short-URL generator (or Key Generation Services):
4. Load Balancers:
5. Cache:
6. Rate Limiter:
7. Propose a Design Diagram and get an agreement

8. Workflow

1. **Shortening**:
2. **Redirection**:
3. **Deletion**:
4. **Custom Short Links**:
5. Specific Design Components:
   * + **Purging or DB Cleanup**:
       - Should
6. Design Evaluation:
7. Availability
   * **Replication of Components**: Most of our building blocks, like databases, caches, and application servers have built-in replication that ensures availability and fault tolerance
   * **Backups to Cloud Storage (Amazon S3)**: To handle disasters, we can perform frequent backups of the storage and application servers, preferably twice a day, as we can’t risk losing URLs data. We can use the Amazon S3 storage service for backups, as it facilitates cross-zonal replicating and restoration as well.
   * **Global Servers Load Balancing (GSLB)**: Our design uses global server load balancing (GSLB) to handle our system traffic. It ensures intelligent request distribution among different global servers, especially in the case of on-site failures.
   * **Rate Limiter**: We also apply a limit on the requests from clients to secure the intrinsic points of failures. To protect the system against DoS attacks, we use rate limiters between the client and web servers to limit each user’s resource allocation. This will ensure a good and smooth traffic influx and mitigate the exploitation of system resources.
8. Scalability

* **Horizontal database sharding**: Our design is scalable because our data can easily be distributed among horizontally sharded databases. We can employ a consistent hashing scheme to balance the load between the application and database layers.
* **Choice of NoSQL database (MongoDB) – supports Horizontal Scaling**: Scaling a traditional relational database horizontally is a daunting process and poses challenges to meeting our scalability requirements. We want to scale and automatically distribute our system’s data across multiple servers. For this requirement, a NoSQL database would best serve our purpose.

1. Readability
   * **Base-58 encoder**: The use of a base-58 encoder, instead of the base-64 encoder, enhances the readability of our system.
2. Latency
   * **Choice of Database (MongoDB)** - Our system is redirection-heavy. Writing on the database is minimal compared to reading. We deliberately chose MongoDB because of its low latency and high throughput in reading-intensive tasks.
   * **Distributed Cache** - The deployment of a distributed cache in our design also ensures that the system redirects the user with the minimum delay possible.
3. Unpredictability

* **Random Selection of Generated Unique IDs**: we can randomly select a unique ID from the available ones and associate it to the long URL, encompassing the unpredictability of our system.

11. Additional Details:

1. Load Balancers:
   * We could use a simple Round Robin approach that distributes incoming requests equally among backend servers.
   * A problem with Round Robin LB is that we do not consider the server load. As a result, if a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that.
2. Caching:

* **Which Cache**: We can use any off-the-shelf solution like Memcached, which can store full URLs with their respective hashes.
* **How much cache memory should we have?** We can start with 20% of daily traffic and, based on clients’ usage patterns, we can adjust how many cache servers we need.
* **Which cache eviction policy would best fit our needs?** When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system.