**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 – 8000
   * URLShortenings/ Sec – 76 URLs/sec
   * URLRedirections/ Sec – 7600 URLs/sec
   * Total Servers Required – 100M/8000 = 12500 servers
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 = 200M/month x 12 months x 5 years = 12Billion
* Total Storage Required = 12B x 500B = 6TB

1. Bandwidth Estimates

* Incoming Data – 76 x 500 x 8 = 304Kbps
* Outgoing Data – 30.4Mbps

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

* Cache Size = ((200M x 500B) / 30) x 0.20 = 66 GB

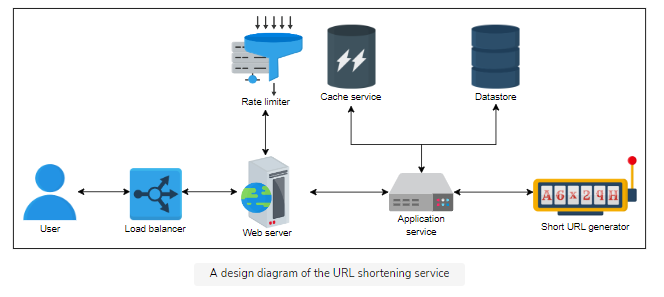
1. System API’s

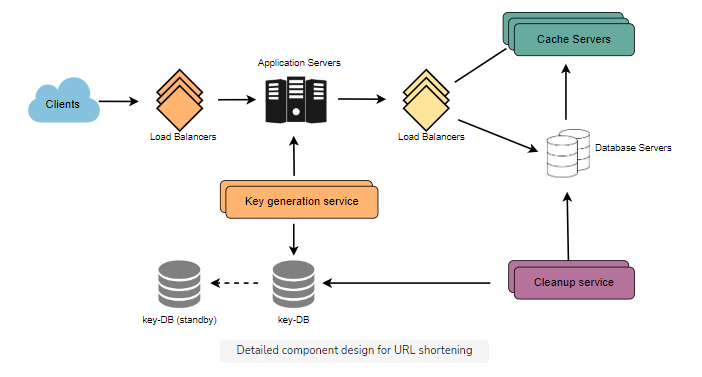
* 3 APIs needed to expose the functionality of our service
  + Shortening a URL - shortURL(api\_dev\_key, original\_url, custom\_alias=None, expiry\_date=None)
  + Redirecting a URL - redirectURL(api\_dev\_key, url\_key)
  + Deleting a URL - deleteURL(api\_dev\_key, url\_key)

1. Database Design

* Our service doesn’t require user registration for the generation of a short URL, so we can skip adding certain data to our database.
* Additionally, the stored records will have no relationships among themselves other than linking the URL-creating user’s details, so we don’t need structured storage for record-keeping.
* Considering the reasons above and the fact that our system will be read-heavy, NoSQL is a suitable choice for storing data. In particular, MongoDB is a good choice for the following reasons:
  + It uses leader-follower protocol, making it possible to use replicas for heavy reading.
  + MongoDB ensures atomicity in concurrent write operations and avoids collisions by returning duplicate-key errors for record-duplication issues.
  + NoSQL databases like Cassandra, Riak, and DynamoDB need read-repair during the reading stage and hence provide slower reads to write performance. They are leader-less NoSQL databases that provide weaker atomicity upon concurrent writes. Being a single leader database, MongoDB provides a higher read throughput as we can either read from the leader replica or follower replicas. The write operations have to pass through the leader replica. It ensures our system’s availability for reading-intensive tasks even in cases where the leader dies.

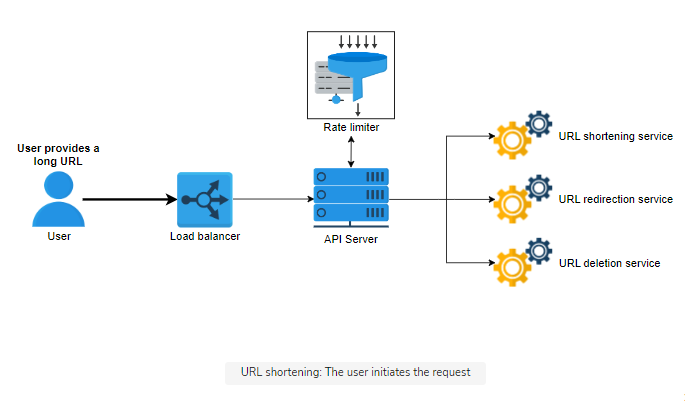
1. Present the building blocks of the Design
2. Short-URL generator (or Key Generation Services): Our short URL generator will comprise a building block and an additional component: 1. A sequencer to generate unique IDs 2. A Base-58 encoder to enhance the readability of the short URL
3. Load Balancers: We can employ Global Server Load Balancing (GSLB) apart from local load balancing to improve availability.
4. Cache: For our specific read-intensive design problem, Memcached is the best choice for a cache solution.
5. Rate Limiter: Limiting each user’s quota is preferable for adding a security layer to our system. We can achieve this by uniquely identifying users through their unique api\_dev\_key and applying one of the discussed rate-limiting algorithms. Keeping in view the simplicity of our system and the requirements, the fixed window counter algorithm would serve the purpose, as we can assign a set number of shortening and redirection operations per api\_dev\_key for a specific timeframe.
6. Propose a Design Diagram and get an agreement

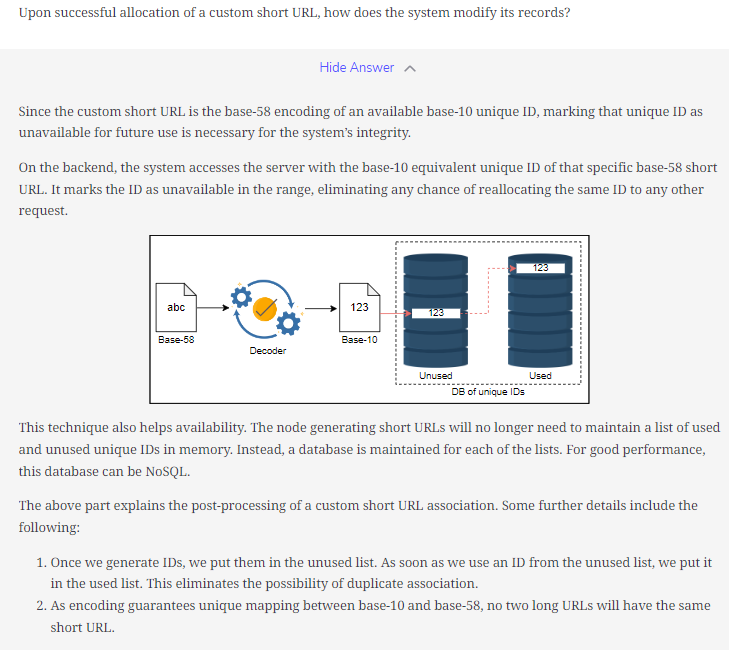




8. Workflow

1. **Shortening**: Each new request for short link computation gets forwarded to the short URL generator (SUG) by the application server. Upon successful generation of the short link, the system sends one copy back to the user and stores the record in the database for future use.
2. **Redirection**: Application servers, upon receiving the redirection requests, check the storage units (caching system and database) for the required record. If found, the application server redirects the user to the associated long URL.
3. **Deletion**: A logged-in user can delete a record by requesting the application server which forwards the user details and the associated URL’s information to the database server for deletion.
4. **Custom Short Links**: Custom short links: This task begins with checking the eligibility of the requested short URL. The maximum length allowed is 11 alphanumeric digits. If the requested URL is available, the user receives a successful short URL generation message, or an error message in the opposite case.





1. Specific Design Components:
   * + **Purging or DB Cleanup**:
       - Should entries stick around forever, or should they be purged? If a user-specified expiration time is reached, what should happen to the link?
       - If we chose to continuously search for expired links to remove them, it would put a lot of pressure on our database. Instead, we can slowly remove expired links and do a lazy cleanup.
       - Whenever a user tries to access an expired link, we can delete the link and return an error to the user.
       - A separate cleanup service can run periodically to remove expired links from our storage and cache. This service should be very lightweight and scheduled to run only when the user traffic is expected to be low.
       - We can have a default expiration time for each link (e.g., two years).
       - After removing an expired link, we can put the key back in the key-DB to be reused.
2. Design Evaluation:
3. 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.
4. 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.