**1.) System Design URL Shortener:**

**Functional Requirements:**

* Given a long URL give short URL
* When user clicks on short URL, redirect to original URL with minimum latency
* User Feed Display
* Comment and like a post

**Non-Functional Requirements:**

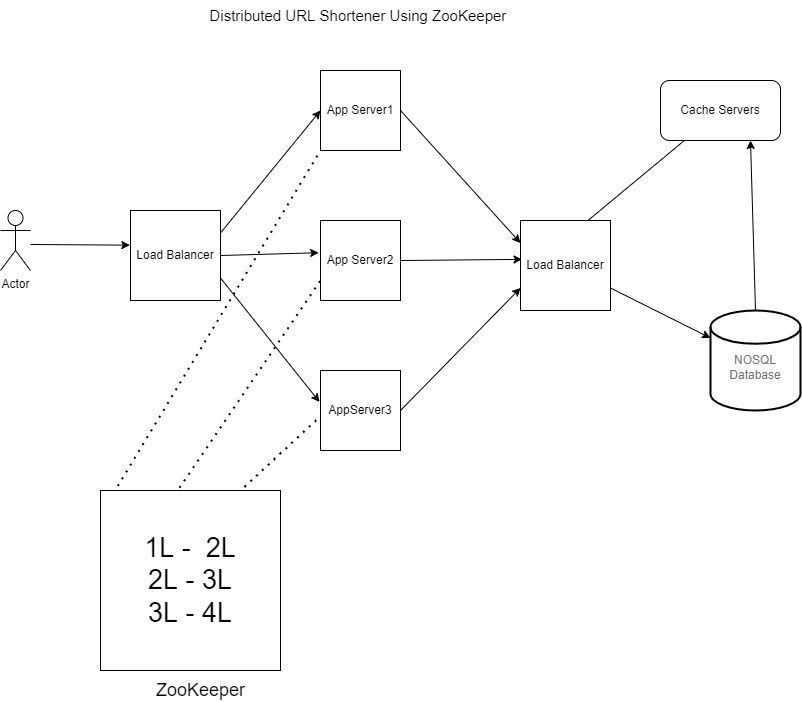
* High Available (Consistency not on priority)
* Scalable

**APIS:**

* createTinyUrl(Long url)
* getLongUrl(Short Url)

**Size of Database Required:**

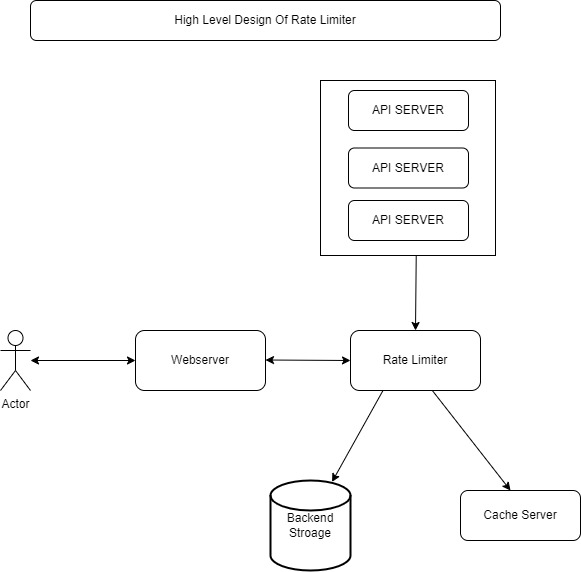
* Each URL mapping size – 2kb
* Number of users per month 10 M – 10M\*2KB = 20GB per month – 1TB 5 years



**2.) Designing an API Rate Limiter:**

**How to implement rate limiter: Most common algorithms are:**

* Token Bucket – 5 ball per min
* Leaky Bucket – Put tasks in queue if queue is full return too many requests return after sometime – (Effective and easy to implement)
* Fixed Window – similar to token bucket – time range is fixed for ex – 5:00 to 05:59 - 5 request – Problems (Clustering around boundary, race condition in case of distributed systems)
* Sliding Window: Better the fixed window : it will check 5 request is previous window if it’s there then it will not accept any new request;



**3.) System Design Instagram:**

**Functional Requirements:**

* Upload Photo/Create Post
* View Photo/Post
* Search Users
* User Feed Display
* Comment and like a post

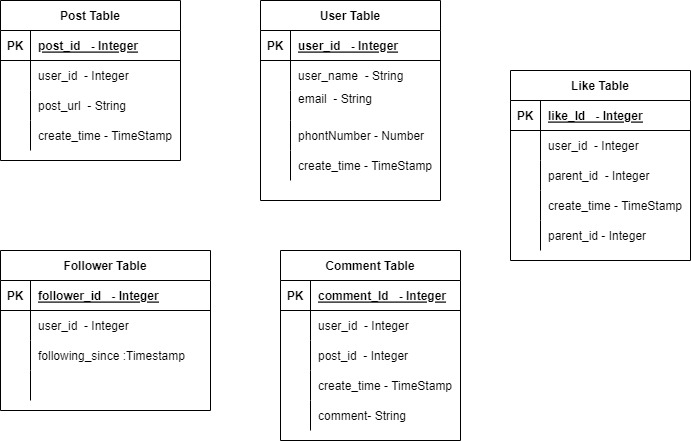
**Non-Functional Requirements:**

* High Available (Consistency not on priority)
* Reliable System hence partition tolerant - photo should not be lost
* Low latency while user feed service
* Read Heavy

Capacity Estimation:

* 100M users - 1M active users - Each Image or post size 1MB
* 1M\*1MB - 975GB perday
* Per year - 975\*365 = 350TB
* sharding will be on post\_id

**Database Schema:**



User Service: getUserDetails, postUserDetails, updateUserDetails

Follow Service: followUser, getListOfUserFollowedByAUser

Post Service: uploadPost, getPost, getUserFeed

Comment And Like Service: addCommentToThePost, getCommentForThePost, addLikeToThePost, getNumberOfLikesForThePost, getListOfUsersWhoLikeThePost

**Promise Based Cache:**

* Cache miss happen for same object for multiple request
* Rather than all request going to db and fetching result and setting in DB
* 1st request creates a promise object in cache
* Subsequent request will cache miss will check if a promise is present in a cache
* If promise is present, all subsequent request will wait for the promise to get completed

