**Designing pb**

1. **Defining pb**
   1. Pastebin is a service which allows users to store plaintext documents in a dynamically generated URL which can be shared and accessed by others
2. **Requirements/Goals**
   1. Functional Requirements
      1. Users should be able to input text data into a frontend interface
      2. Users should be able to generate a unique URL to share/access the data
      3. Users should be able to signup/login and view all of their previously created text data documents/URLs
      4. Text data should expire after a specific timespan automatically. Users should also be able to delete text data / URLs associated with their account immediately
   2. Non-Functional Requirements
      1. The system should be highly available
      2. The system should be very fast with minimal latency
      3. The system should be highly reliable, and data should not be lost
      4. Generated URLs should be randomized/hashed as to prevent guessing URLs
   3. Other Requirements
      1. Analytics (# of views, view access location)
3. **Design Considerations**
   1. URL shortening service incorporated into requirements
   2. File size limit for user text data (10MB)
4. **Capacity Estimation/Constraints**
   1. Assume a 5:1 read to write ratio
   2. Traffic estimates will be lower overall than compared to bit.ly / Instagram / FB
      1. We can estimate 1M new pastes added to our DB each day
   3. **Bandwidth estimates:**
      1. **Read**: 58 rps => 580 KB/s
      2. **Write**: 12 rps => 120 KB/s
   4. **Capacity estimates:** 
      1. 1M (pastes / day) \* 10MB (max size of single paste) => 10 GB / day
      2. Estimate capacity for 5 year storage => 3.6TB \* 5 => **18TB** (5 year capacity)
   5. **URL estimates**
      1. 1M new pastes / day \* (5 \* 365) => **1.8 Billion unique URLs needed**
      2. Base64 \* 6 digit => 68.7B unique string combinations
      3. Base64 \* 8 digit => 281 Trillion unique string combinations
      4. Based on our maximum estimates for traffic volume (write), base64 \* 6 digit should suffice for our requirements
   6. **Cache estimates**
      1. If we follow 80:20 rule, where 80% of the most frequently accessed URLs are generated from the top 20% of URLs, then we can assume that caching the top 20% will allow us to streamline our service reducing latency for users
      2. 5 Mil read requests / day \* 10MB \* 0.2 => ~= **10GB**
5. **API**
   1. POST (/users)
      1. Requires user login / paste data / paste name / URL generation
   2. GET (/users)
   3. POST (/pastes)
   4. GET (/pastes)
   5. PUT (/pastes)
6. **Schema/DB Design**
   1. **Users**
      1. Id: primary key
      2. Username/email: varchar(255)
      3. Password\_digest: varchar(255)
      4. Date\_joined: timestamp
   2. **Pastes:**
      1. Id: primary key
      2. Paste\_Data: textarea (Max: 10MB)
      3. Paste\_title: varchar(255)
      4. Author\_id: foreign key
      5. Meta\_data: varchar(255)/Integer
      6. Date\_created; Timestamp
      7. Visit\_count: Integer/BigInt
7. **Design**
   1. Client hits load balancer on request. Request load is distributed across multiple web servers
   2. Web servers then pass request to load balanced application servers
   3. Depending on the nature of the request, different data flow request/response will occur
      1. Write (POST):
         1. Client makes request to POST new paste
            1. Request is sent to application server
            2. Key generation service produces unique URL
            3. POST request is sent to relational DB to store metadata
            4. Object is sent to object-storage (e.g. s3, cassandra)
            5. Upon successful entry into database, the response is sent back up to the client. The response will include the newly generated link to the user/client
         2. Client makes GET request to retrieve existing paste
            1. GET Request is sent to servers
            2. Metadata is retrieved from relational DB store for specific paste entry
            3. Object-storage is queried, returning the paste entry
            4. Any relevant metadata is updated
            5. The entry/text data is returned back to the client