9.8

Design a system like Pastebin

What is Pastebin?

Allows users to store plain text over the internet and provide unique URLs to access the text

Often used to quickly share source code, configs or logs. Since users just need to share the URL to let others see it.

Functional Requirements

1. Users should be able to upload /“paste” their text and get a unique URL to access it.
2. Users can delete their pastes.
3. Content and URLs will automatically expire after a specific time span. Also allow users to specify expiration time.

Non Functional Requirements

1. Highly reliable, uploaded Pastes should not be lost.
2. Highly available. If the service is down, users will not be able to access their Pastes.
3. Fast. Users should be able to access their Pastes in real-time with minimum latency.
4. Paste links should not be predictable (Do NOT use sequential ordering).

Storage Estimation for Pastes

Let’s make the max size of a paste 10MB to prevent abuse, but we expect an average paste size of **10KB.**

Let’s assume we have 10 million users, and each user submits 1 paste (1 write request) a month.

This means we need a total storage capacity of 11.72 TB to store the pastes for 10 years. (1.2B Pastes * 10KB)

Storage Estimation for URLs

With 10 million pastes a month, this will be 1.2B Pastes over 10 years. Meaning we will need 1.2B Unique URLs over 10 years.

We will use Base64 encoding for our unique URL (([A-Z, a-z, 0-9, ., -])).

Lets make our URL be 6 characters long since 68.7B (64 ^ 6) possible unique URLs should be more than enough.

It takes one byte to store one character, total size required to store the Unique URLs would be ~7GB (1.2B * 6) for 10 years.

Storage Estimation for Pastes + URLs

We need a total storage capacity of 11.72TB to store the pastes for 10 years. (1.2B Pastes * 10KB)

It takes one byte to store one character, total size required to store the Unique URLs would be ~7GB (1.2B * 6) for 10 years.

11.72TB + 7GB

Storage Choice and Design

We can separate our storage layer by using an Object Storage Service, such as Amazon S3, for storing the paste content and a NoSQL database for storing URLs.

This separation will also allow us to scale them individually.

Todo: Insert diagram here

Generating Unique URL

When generating a new unique URL, we could have our server generate a random 6 letter string. This would work, but there'd be a small chance (1 / 68.7B) that duplicate 6 letter strings are generated.

In fact as we create more unique URLS, the chances of generating one that already exists will increase!

A simple solution would be to have our server keep generating a new key until one is found that does not already exist...

Generating Unique URL

Another solution to this problem would be to pre-generate the 68.7B 6 letter strings beforehand using a **Key Generation Service (KGS)**, and then hand out those pre-generated strings when needed.

We will no longer have to worry about duplications or collisions!

But we would have to worry about a single point of failure, the **Key DB** that stores the 68.7B 6 letter strings pre-generated from the Key Generation Service (KGS).

Make sure you have a backup KGS and Key DB.

Traffic + Bandwidth Estimation

Traffic:

At 10M Pastes a month, this averages out to ~3.8 Pastes per second.

Let’s make a broad assumption that each paste is seen by an average of 10 people. This averages out to ~38 Reads per second.

Bandwidth:

For pastes (write requests), we therefore expect a ingress of 38 KB/S (3.8 * 10KB)

For read requests, we therefore expect a egress to users of 0.38 MB/S (38 * 10KB)

API Requirements

Let’s use either a SOAP or REST API when implementing our system.

createPaste(api_key, paste_content, user_name, paste_name, expire_date)
Returns paste URL or error depending on success.

readPaste(url)
Return paste content or error depending on validity of paste URL.

deletePaste(api_key, url)
Return boolean of true/false depending on success

Database Design

1. Pastebin will be read heavy (10 to 1 Read/Write Ratio).
2. Only one relationship between records, the relationship being which user created which paste.

For storing the paste content, we have to choose between a SQL vs. NoSQL Database.

NoSQL databases are not as good at looking at relationships, in exchange they're faster for writes and simple key-value reads.

Database Design

User Table	
Primary Key	User ID
	Name
	Email
	Password
	Creation Date

Paste Table	
Primary Key	User ID
	Short URL
	Paste Path
	Expiration Date
	Visits

Paste Path: The URL of Amazon S3 Bucket, where the paste content is stored.

Database Clean Up

“Content and URLs will automatically expire after a specific time span. Also allow users to specify expiration time”

Let’s run a cleanup service to remove expired links from our storage and cache. This service should be lightweight and scheduled for times when user traffic is low.

After removing an expired URL, we can put the key back in the key-DB to be reused.

Return errors if user tries to access an expired URL/paste.

Caching

We know that our service will be read heavy (10:1). We can speed up the reading process using a cache.

Let’s start off by putting 20% of our daily pastes into the cache. The 80-20 rule would say that 80% of traffic is driven by 20% of the pastes.

When an application requests a paste, it first tries the cache before checking the backend. Thus allowing faster access / lower delay.

Whenever the cache is full, we need to replace the URLs with a newer/hotter URL. Let’s use a Least Recently Used (LRU) cache eviction policy.

Load Balancing

Takes loads which are requests from users, and it routes traffic to different back end servers.

Reliability and high availability is maintained by redirecting requests only to servers which are available.

Let’s add a Load balancing Layer at these places in our system:

1. Between Clients and Application servers

2. Between Application Servers and database servers

Load Balancing

We could start with a Round Robin approach since it is easy to implement and does not introduce any overhead.

Round Robin LB does not consider the server load.

A better LB solution could determine where to route based on Least Connection or Least Response or Least Bandwidth, etc.

Let’s go with Least Bandwidth.

Insert final diagram here