Design Pastebin: System Design Interview Question

Asked In: Amazon, Ola Cabs

What is Pastebin?

Have you ever wanted to store data online and make it accessible to others? If so, then you may be interested in the Pastebin system design. Pastebin is a service that lets users post various types of content, such as text and images, on the internet and share them with others using a unique URL. The creator of the content can also update it if they are logged in.

Key Requirements

Functional Requirements

- Users should be able to generate a unique URL by pasting content into the system.
- The content and URLs should expire after a specified amount of time

Non-Functional Requirements

- System should be reliable and have high availability.
- System should be available in real-time with minimal latency.
- Paste URL should not be predictable.

As we know what our essential requirements are, it’s time to estimate capacity of the system.

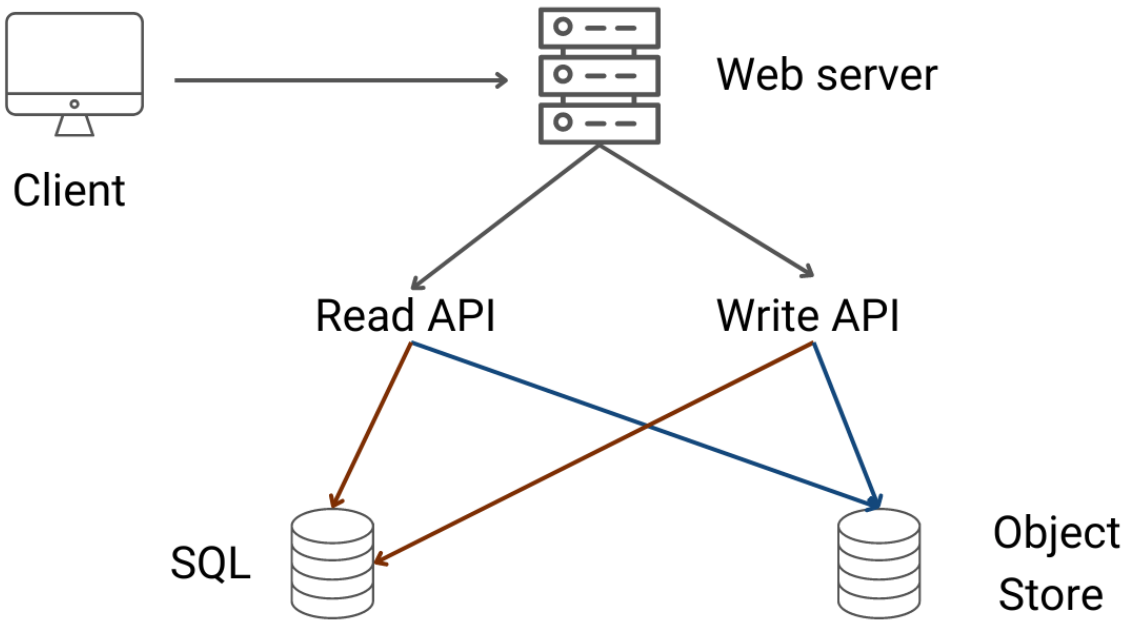
Capacity Estimation

Since this system is expected to receive more reads than writes, let's consider a read-to-write ratio of 10:1. Suppose we have 10 million users,

and each user makes one write request per month on average. This means we can expect the following traffic:

- 10 million write requests per month.
- 100 million read requests per month (10 times the number of write requests).
- 1.2 billion write requests over 10 years (10 million per month * 12 months/year * 10 years).
- Suppose the maximum size per paste is 10 MB, resulting in 100 TB of paste per month (10 million pastes * 10 MB/paste).
- 1.2 PB of paste content over a year (100 TB/month * 12 months/year).
- Total 12PB of data will be accumulated over 10 years (1.2 PB/year * 10 years).

High Level Design



In the Pastebin system, we will have two main APIs: One for writing content and another for handling reading requests. These APIs can be implemented using either the SOAP or REST architecture.

Write API: write(apiKey, content, expiryDate)

- **apiKey:** A unique API key associated with each user.
- **content:** The user provides the content that they want to paste.
- **expiryDate:** The expiry date indicates the time after which the content should expire.
- The API returns a short URL that can be used to access the content. If the input is invalid, an error code will be returned.

To create a paste, the client sends a request to the web server, which forwards it to the write API server. The write API server then performs the following actions:

- Generates a unique URL.
- Stores the URL in the pastes table of the SQL database.
- Stores the content in an object store and returns the URL.

Read API: read (shortUrl)

- **shortUrl:** The URL is provided by the write API to access the content.
- The API returns the original paste content or an error code if the URL is incorrect.

To view a paste, the client sends a request to the web server, which forwards it to the read API server. The read API server then performs the following actions:

- Checks the SQL database for the generated URL.
- Retrieves the paste content from the object store if the URL is present in the database.
- Returns an error message to the user if the URL is not found.

Delete API: delete (apiKey, URL)

- **apiKey:** A unique API key provided to each user.
- **URL:** The URL of the content that the user wants to delete.
- Delete API returns a boolean value indicating the success or failure of the operation.

In addition to these essential services, we also have two additional APIs that handle real-time analytics and the deletion of expired content.

Database Design

Both the content and short URL need to be stored. The average size of each content is 10 MB. As previously calculated, our service will need to store a total of 12 PB of data over the course of 10 years, which is too much to store in a traditional database. Therefore, we will use an object storage service, such as Amazon S3, to store the content and a relational database to store the URLs.

Database Structure

The database has the following structure:

User Data

1. User ID: A unique identifier for each user.
2. Name: The name of the user.
3. Email: The email address of the user.
4. Password: The password of the user.
5. Creation Date: The date on which the content was shared.

Content Data

1. User ID: A unique identifier for each user.
2. Short URL: A unique, 7-character short URL.
3. Paste path: The URL path of the S3 bucket.
4. Expiration date: The time after which the content expires.
5. Last accessed: The last time the content was accessed.

The User ID serves as the primary key in this structure.

User Data	
Attribute	Type
User ID (Primary key)	Varchar
Name	Varchar
Password	Varchar
Email	Varchar
Creation Date	DateTime

Content Data	
Attribute	Type
User ID (Primary key)	Varchar
Short URL	Varchar
Paste Path	Varchar
Expiration Date	DateTime
Last Accessed	DateTime

For storing paste URLs, we have two options: a relational database like MySQL or a NoSQL database. Since we need fast read and write speeds but do not have many dependencies or relationships among our data, a NoSQL database (like Key-Value Store) is the optimal choice for our system.

Relational databases are very efficient when there are many dependencies and complex queries, but they can be slow for our purposes. On the other hand, NoSQL databases may not be as good at handling relationship queries, but they are faster overall.

Low Level Design

The key components of the pastebin system are generating short URLs and storing content. Let's dive into these topics in more detail.

Generating Short URL

Every time a user creates a paste, the system needs to generate a unique, short URL. For this, we can use BASE64 encoding because all BASE64 characters are URL-safe (i.e., they are within the range of [A-Z], [a-z], [0-9], '+', and '/').

- To generate a unique URL, we can use the MD5 hash algorithm. MD5 is a cryptographic hash function algorithm that takes input data of any length and produces a fixed-size output of 128 bits or 16 bytes.
- We take the MD5 hash of the user's IP address and timestamp (or randomly generated data) to create a unique URL for the paste. Now we encode the MD5 result using BASE64, which will be 22-character string. Note: Base64 encoding will use 6 bits to represent each character. So Base64 encoding of MD5 output will give 128/6 i.e. ~ 22 character output.
- Now for choosing the short URL, we can choose the first 6 characters of the resulting Base64 encoding.

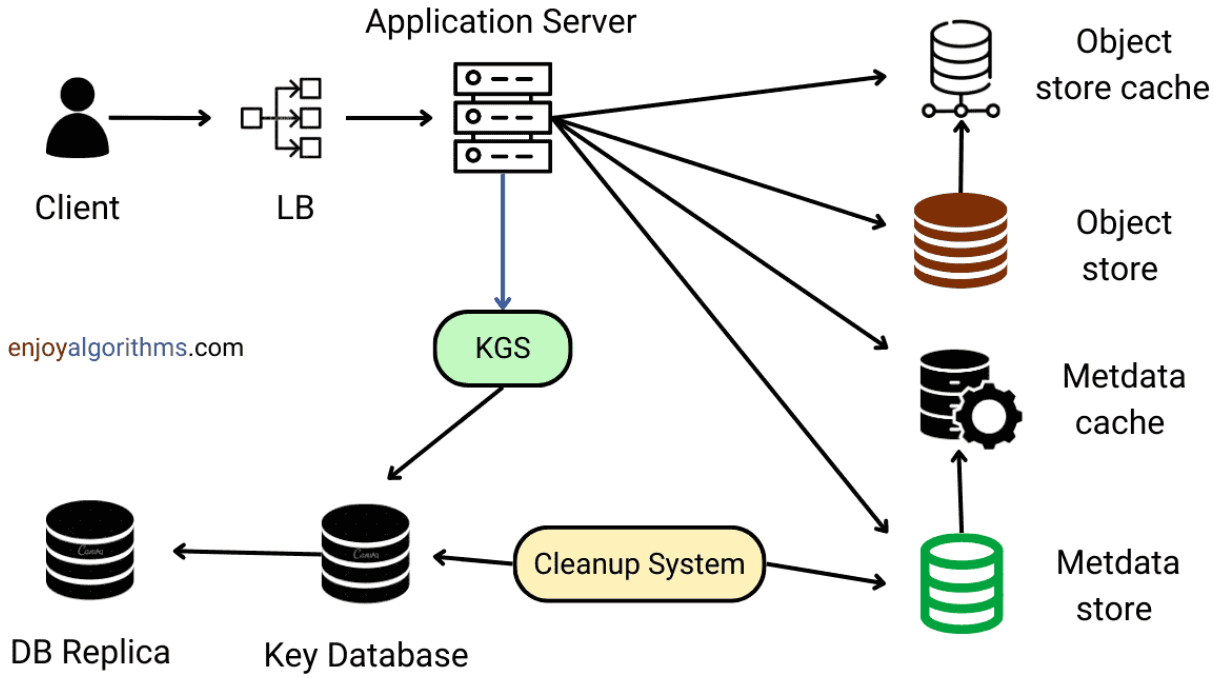
The critical idea to explore: What is the MD5 hash algorithm? How does it work? What are its use cases in system design?

The drawback of the above approach

One issue with this algorithm is that the generated URL may be repeated. In such a case, we need to regenerate a new key and keep trying until we don't encounter any failure due to the duplicate key. To address this, we can use a Key Generating Service (KGS) to ensure that all keys inserted into the key database are unique and prevent collisions or duplications.

Key Generation Service (KGS) generates random six-letter strings beforehand and stores them in some database. Whenever we want to store a new paste, we can select an already generated key from the DB and use it. KGS will also ensure that all keys inserted into DB are unique.

- KGS can use two tables to store keys: One for unused keys and one for all used keys. As soon as KGS provides keys to an application server, it can move them to the used keys table, ensuring that each server gets unique keys.
- We can have a standby replica of KGS to prevent a single point of failure. This replica can take over and generate and provide keys whenever the primary server fails.



Scaling the System

Scalability, reliability, availability, and performance are critical business requirements for this service. It should be able to handle a large number of requests and provide a fast, seamless experience for users. To achieve this, we will need to scale our database and have additional servers to handle the increased traffic.

Database Sharding

If we use a single database to store all of the data, the service will be more prone to failure. To prevent this, we can partition the data across multiple machines or nodes to store the billions of URLs.

To partition the database, we can use a technique called hash-based partitioning. This involves using a hash function to distribute the URLs into different partitions. We need to decide how many shards to create and choose an appropriate hash function that maps each URL to a specific partition or shard number.

Caching

Since we have ten times more read operations than write operations, we

can use caching to speed up the service. Caching involves storing frequently accessed URLs in memory for faster access. For example, if a URL appears on a trending page on a social networking website, it is likely to be visited by many people. In this case, we can store the content of the URL in our cache to avoid delays. Services like Memcached or Redis can be used for this purpose.

There are several considerations to keep in mind when designing the cache:

- When the cache is full, we need to replace less frequently used URLs with more popular ones. We can use the Least Recently Used (LRU) method to implement this.
- The cache should be kept in sync with the original content. If any changes are made to the original content, they should be reflected in the cache.

We can use a distributed cache to address these issues.

Load Balancing

There may be instances where a single server receives a large number of requests, causing the service to fail. To prevent this single point of failure, we can use load balancing.

There are many algorithms available for distributing the load across servers, but in this system, we can use the least bandwidth method. This algorithm directs incoming requests to the server with the least amount of traffic.

In addition to load balancing, we also need to replicate all databases and servers to ensure that the service remains highly consistent and does not go down in case of any discrepancies.

Conclusion

Pastebin system is a complex and highly scalable service to design. In this