Designing Pastebin

Difficulty Level: Easy

Let’s design a Pastebin-like web service, where users can store plaintext.

Users of the service will enter a piece of text, and get a randomly generated URL to access it.

1. What is Pastebin?

Pastebin-like services enable users to store plaintext or images over a network (typically the Internet) and generate unique URLs to access the uploaded data.

Such services are also used to share data over the network quickly, as users would just need to pass the URL to let other users see it.

2. Requirements and Goals of the System

Our Pastebin service should meet the following requirements:

Functional Requirements

1. Users should be able to upload or “paste” their data and get a unique URL to access it
2. Users will only be able to upload text
3. Data and links will expire after a specific timespan automatically; users should also be able to specify expiration time
4. Users should optionally be able to pick a custom alias for their paste

Non-Functional Requirements

1. The system should be highly reliable, any data uploaded should not be lost
2. The system should be highly available — this is required because if our service is down, users will not be able to access their pastes
3. Users should be able to access their pastes in real-time with minimum latency

4. Paste links should not be guessable (not predictable)

Extended Requirements

1. Analytics, e.g. how many times was a paste accessed?
2. Our service should also be accessible through REST APIs by other services

3. Some Design Considerations

Pastebin shares some requirements with *URL Shortening Service*, but there are some additional design considerations we should keep in mind.

What should be the limit on the amount of text the user can paste at a time?

We can limit users not to have pastes bigger than 10MB to stop the abuse of the service.

Should we impose size limits on custom URLs?

Since our service supports custom URLs, users can pick any URL that they like, but providing a custom URL is not mandatory.

However, it is reasonable and desirable to impose a size limit on custom URLs so that we have a consistent URL database.

4. Capacity Estimation and Constraints

Our services will be read-heavy; there will be more read requests compared to new Paste creation.

We can assume a 5:1 ratio between the read and write.

Traffic Estimates: Pastebin services are not expected to have traffic similar to Twitter or Facebook. Let's assume here that we get 1 million new pastes added to our system every day. This leave us with 5 million reads per day.

New pastes per second:

$$\frac{1M}{24 \text{ hours} \cdot 3600 \text{ seconds}} \approx 12 \text{ pastes/sec}$$

Paste reads per second:

$$\frac{5M}{24\text{ hours} \cdot 3600\text{ seconds}} \approx 58\text{ reads/sec}$$

Storage Estimates: Users can upload a maximum 10MB of data. Commonly, Pastebin-like services are used to share source code, configs, or logs. Such texts are not huge, so let’s assume that each paste on average contains 10KB.

At this rate, we will be adding 10GB of data per day:

$$1M \cdot 10\text{ KB} = 10\text{ GB/day}$$

If we want to store this data for ten years, we would need a total storage capacity of 36TB.

With 1 million pastes every day, we will have 3.6 billion pastes in 10 years.

We need to generate and store keys to uniquely identify these pastes.

If we use Base64 encoding ([a-zA-Z0-9+/]), we would need 6 letter strings:

$$64^6 \approx 68.7\text{ billion unique strings}$$

If it takes one byte to store one character, total size required to store 3.6 billion keys would be:

$$3.6B \cdot 6 = 22\text{ GB}$$

22GB is negligible compared to 36TB.

To keep some margin, we will assume a 70% capacity model (meaning that we don’t want to use more than 70% of our total storage capacity at any point), which raises our storage needs to 51.4TB.

Bandwidth Estimates: For write requests, we expect 12 new pastes per second, resulting in 120KB of ingress per second.

$$12 \cdot 10\text{ KB} = 120\text{ KB}$$

As for the read requests, we expect 58 requests per second. Therefore, total data egress (i.e. data sent to users) will be 0.6 MB/s.

$$58 \cdot 10\text{ KB} = 0.6\text{ MB}$$

Although total ingress and egress are not that large, we should keep these numbers in mind while designing our service.

Memory Estimates: We can cache some of the hot pastes that are frequently accessed. Following the 80-20 rule, meaning 20% of hot pastes generate 80% of the traffic, we would like to cache these 20% pastes.

Since we have 5 million read requests per day, to cache 20% of these requests we would need:

$$0.2 \cdot 5M \cdot 10 \text{ KB} \approx 10 \text{ GB}$$

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service.

The following could be the definitions of the APIs to create, retrieve, and delete pastes:

```
addPaste(api_dev_key, paste_data, custom_url=None user_name=None, paste_name=None, expire_d
ate=None)
```

Parameters:

- `api_dev_key` (string) — the API developer key of a registered account; this will be used to, among other things, throttle users based on their allocated quota
- `paste_data` (string) — textual data of the paste
- `custom_url` (string) — optional custom URL for the paste
- `user_name` (string) — optional user name to be used to generate the URL
- `paste_name` (string) — optional name of the paste
- `expire_date` (string) — optional expiration date for the paste

Returns: (string)

A successful addition returns the URL through which the paste can be accessed, otherwise, it will return an error code.

```
getPaste(api_dev_key, api_paste_key)
```

Parameters:

- `api_dev_key` (string) — same as above
- `api_paste_key` (string) — represents the paste key of the paste to be retrieved

Returns: (string)

A successful retrieval returns the textual data of the paste, otherwise, it will return an error code.

```
deletePaste(api_dev_key, api_paste_key)
```

Parameters:

- `api_dev_key` (string) — same as above
- `api_paste_key` (string) — same as above

Returns: (Boolean or string)

A successful deletion will return some kind of success message, otherwise it will return an error code.

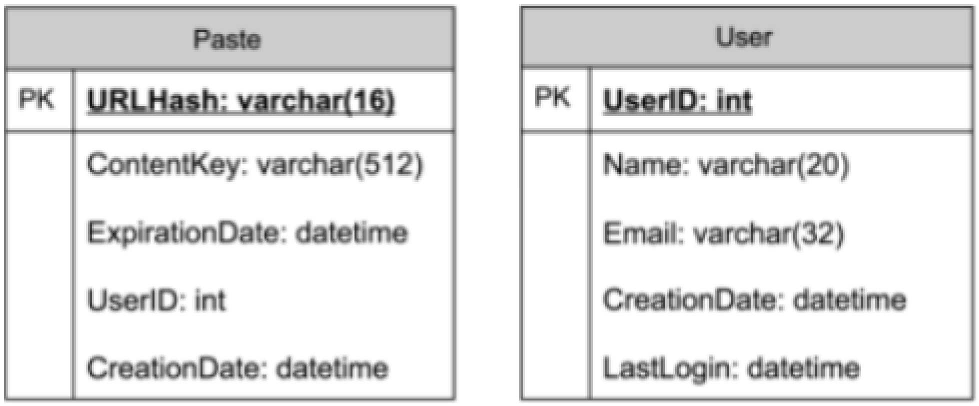
6. Database Design

A few observations about the nature of the data we are storing:

1. We need to store billions of records
2. Each metadata object we are storing would be small)less that 1KB)
3. Each paste object we are storing can be of medium size (it can be a few MB)
4. There are no relationships between records, unless we also want to store which user created what pastes
5. Our service is read-heavy

Database Schema

We would need two tables: one for storing information about the pastes, and the other for users' data.



Paste	
PK	**URLHash: varchar(16)**
	ContentKey: varchar(512)
	ExpirationDate: datetime
	UserID: int
	CreationDate: datetime

User	
PK	**UserID: int**
	Name: varchar(20)
	Email: varchar(32)
	CreationDate: datetime
	LastLogin: datetime

Here, ‘URLHash’ is the URL equivalent of the TinyURL, and ‘ContentKey’ is a reference to an external object storing the contents of the paste.

We’ll discuss the external storage of the paste contents shortly.

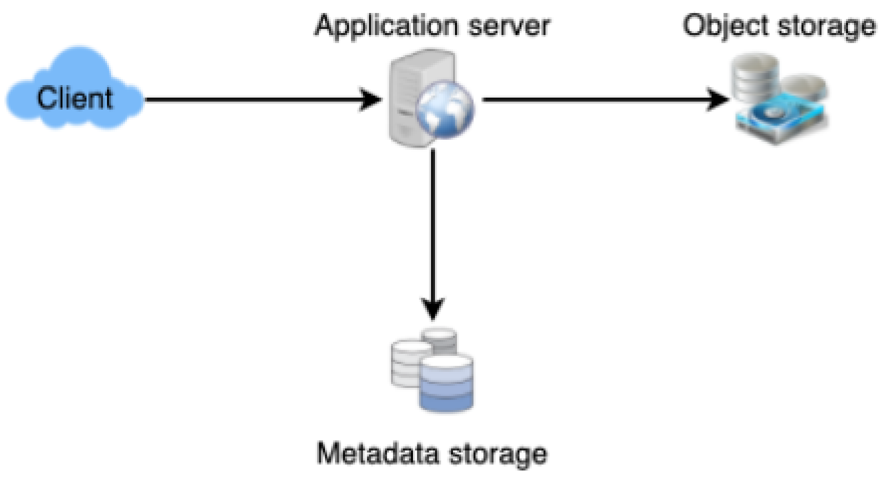
7. High-Level Design

At a high-level, we need an application layer that will serve all the read and write requests.

The application layer will talk to a storage layer to store and retrieve data.

We can segregate our storage layer with one database storing metadata related to each paste, users etc., while the other database stores the paste contents in some object storage.

This division of data will also allow us to scale them individually.



High level design of Pastebin

8. Component Design

a. Application Layer

Our application layer will process all incoming and outgoing requests. The application servers will be talking to the backend data store components to serve the requests.

Handling Write Requests

Upon receiving a write-request, our application server will generate a 6 letter random string, which serves as the key of the paste (if the user does not provide a custom key).

The application server will then store the contents of the paste and the generated key in the database.

After successful insertion, the server can return the key to the user.

One possible problem here could be that the insertion fails because of a duplicate key.

Since we are generating a random key, there is a possibility that the newly generated key could match an existing one.

In this case, we should regenerate a new key and try again. We should keep retrying until we don't see failure due to the duplicate key.

We should return an error to the user if the custom key they have provided is already present in our database.

Another solution for the above problem could be to run a standalone Key Generation Service (KGS) that generates random 6 letter strings beforehand and stores them in a database — let's call it the key-DB.

Whenever we want to store a new paste, we will just take one of the already generated keys and use it.

This approach will make things quite simple and fast since we will not need to worry about duplications or collisions. KGS will make sure that all the keys inserted into key-DB are unique.

KGS can use two tables to store keys: one for keys that have not yet been used, and one for all the used keys.

KGS can also always keep some keys in memory so that whenever our application server needs them, they can be quickly provided.

As soon as KGS loads some keys into memory, it can move them to the used keys table; this way we can make sure that each server gets unique keys.

If KGS dies before using all the keys loaded in memory, we will be wasting those keys. But we can ignore these keys given that we have a huge number of them.

Isn't KGS a single point of failure?

Yes, it is.

To solve this, we can have a standby replica of KGS, and whenever the primary server dies it can take over to generate and provide keys.

Can each application server cache some keys from key-DB?

Yes, this can surely speed things up.

Although in this case, if the application server dies before consuming all the keys, we will end up losing those keys.

This could be acceptable given that we have 68 billion unique 6 letter keys, which is a lot more than we require.

Handling Read Requests

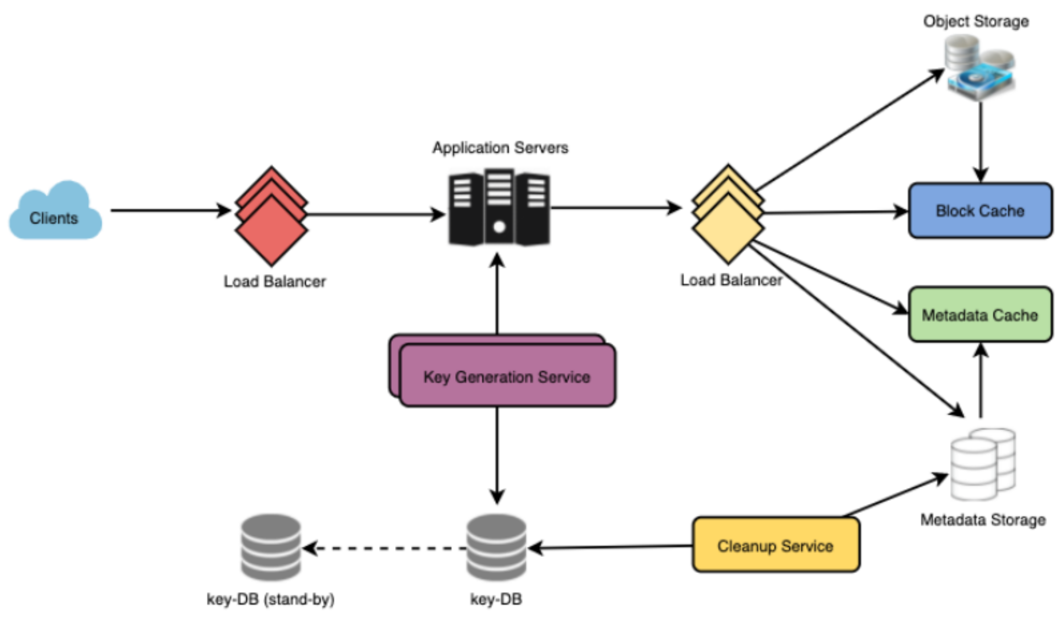
Upon receiving a read paste request, the application service layer contacts the datastore.

The datastore searches for the key, and if it is found, it returns the paste's contents. Otherwise, an error code is returned.

b. Datastore Layer

We can divide our datastore layer into two:

1. Metadata Database — we can use a relational database like MySQL, or a distributed Key-Value store like Dynamo or Cassandra
2. Object Storage — we can store our contents in an object storage (like Amazon's S3); whenever we feel like we are hitting our full capacity on content storage, we can easily increase it by adding more servers



Detailed component design for Pastebin

9. Purging or DB Cleanup

See *Designing a URL Shortening Service like TinyURL*.

10. Data Partitioning and Replication

See *Designing a URL Shortening Service like TinyURL*.

11. Cache and Load Balancer

See *Designing a URL Shortening Service like TinyURL*.

12. Security and Permissions

See *Designing a URL Shortening Service like TinyURL*.

13. Telemetry

See *Designing a URL Shortening Service like TinyURL*.