**URL Shortener | TinyURL | System Design Interview Question**

**Use case**

URL Shorteners are widely used on the Internet. One of the most popular services is [TinyURL](https://tinyurl.com/). It can generate a short URL [tinyurl.com/algomo](https://tinyurl.com/algomo) that redirects to [algo.monster](https://algo.monster/). And when you post a link on Twitter, it will automatically generate a short URL, such as [t.co/tfIrCKKUrd](https://t.co/tfIrCKKUrd). You can read more about how URL redirection works on [MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections).

We will design a scalable URL shortening service with [real demo code](https://github.com/realAlgoMonster/system-design/tree/main/tiny-url-shortener).

**API**

For simplicity, our demo will not have user authentication and all requests are made anonymously.

* GET /<short> 302 redirects to the long URL if the short id is valid, otherwise return 404 not found.
* POST /api/link request: longURL is the long URL to be shortened. response: short is the short id encoded in Base58. tinyurl is the shortened URL. Both request and response are in JSON.

In practice, we often need to support additional fields like custom\_alias and expiry\_date.

**Back of the Envelope Calculation**

Let's do some estimation and calculation. Suppose we have 1 million DAU(Daily Active User) and each user make 3 shortening requests per day. Then there are 3 million new links per day and about 1 billion per year. We have 1000000/86400≈12 requests per second on average. Suppose the peak traffic is 5 times of the average, our system should be able to handle 60 shortening requests per second at peak. Suppose the read to write ratio is 10:1, we need to handle 600 redirection requests at peak.

We also need to determine the length of the short id, which should be as short as possible. Assume we have an alphabet of 58 characters (reason in the implementation section) for the short id, here is a table about the number of distinct short id of the given length.

| **Length** | **Number of distinct short ID** |
| --- | --- |
| 1 | 58^1 = 58 |
| 2 | 58^2 = 3,364 |
| 3 | 58^3 = 195,112 |
| 4 | 58^4 = 11,316,496 |
| 5 | 58^5 = 656,356,768 |
| 6 | 58^6 = 38,068,692,544 |
| 7 | 58^7 = 2,207,984,167,552 |

As from this table, there are 38 billion distinct short IDs of length 6 in Base58 and 2 trillion of length 7. Although length 6 is enough for more than 30 years, we will use length 7 in our demo to avoid possible collisions.

As for storage, assume each long URL is 100 bytes on average, then we can store about 10 million links in 1GB storage. Hence we need about 100GB storage per year.

**Service**

Our example is very simple so there is only one service that serves the two API endpoint. In practice, we also need an User service to handle authentication.

Here are two diagrams to illustrate the idea.

**Storage**

Our data model for now is very simple and there is no complex relationship. We just need a map from short id to long URL, and we have much more reads than writes. So a key-value NoSQL database is most suitable in this case, and our choice is [Redis](https://redis.io/). Redis is widely used as a key-value cache, but it's also a distributed key-value database with on-disk persistence with [Redis cluster](https://redis.io/topics/cluster-tutorial). Redis is easy to use in python with [redis-py](https://github.com/andymccurdy/redis-py) and easy to deploy with [docker](https://hub.docker.com/_/redis) for early development.

To minimize the chance of data loss, we need to use the AOF Redis. You can read more about [Redis Persistence](https://redis.io/topics/persistence).

Redis is not perfect. Using Redis as a database sacrifices write performance to achieve persistence. Moreover, we must have enough memory to store all the data, so we might run out of memory as the service gets more popular. But from our calculation, we can store about 10 million links in 1GB memory, and we can split the dataset among multiple nodes with the Redis cluster, so memory is not a big problem(at least in the first year). Finally, even if we use other distributed key-value databases, it's very common to use Redis as a cache layer.

**Detailed Implementation**

**Short code**

For each shortening request with a long URL, we need to assign a short code and store the pair in our database, then redirects <BaseURL>/<short> to the long URL. There are various ways to generate the short code. Using an auto increment id is the easiest, but people can guess the id of other links, which is insecure if we want to keep the long URL confidential to only people who know the short link. We can also calculate the hash of the long URL and use truncate the hash as short id, but then if two users shorten the same long URL, the short id will be the same, which is not ideal if we want to add analytics in the future. We can solve this problem by adding a counter or random salt to the end of the long URL before hashing. Another approach that prevents these problems is to randomly generated an short id. We can use integer for id in our database and encode it to a short id in the short URL.

**Encoding and decoding**

Base64 is commonly used to present binary data in URLs. We can use Base58 is our case, which removed 0OIl+/ from the alphabet to avoid visually identical looking. It's very easy to implement Base58 in less than 30 lines in python.

1# utils.py

2class Base:

3 alphabet: str

4 size: int

5 decode\_map: dict

6

7 def \_\_init\_\_(self, alphabet: str):

8 self.alphabet = alphabet

9 self.size = len(alphabet)

10 self.decode\_map = dict()

11 for i in range(self.size):

12 self.decode\_map[alphabet[i]] = i

13

14 def encode\_int(self, i: int) -> str:

15 out = ''

16 while i:

17 i, idx = divmod(i, self.size)

18 out = self.alphabet[idx] + out

19 return out

20

21 def decode\_int(self, data: bytes) -> int:

22 i = 0

23 for char in data:

24 i = i \* self.size + self.decode\_map[char]

25 return i

26

27BASE58 = Base('123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz')

28

**Data**

1# models.py

2import redis

3import random

4import json

5from types import SimpleNamespace

6

7BASE = 58

8LENGTH = 7

9

10r = redis.Redis(host='localhost', port=6379, db=0)

11

12class Link:

13 id: int

14 longURL: str

15

16 def \_\_init\_\_(self, longURL: str):

17 self.longURL = longURL

18

19 @classmethod

20 def from\_redis(cls, id: int):

21 data = r.get(id)

22 if not data:

23 return None

24 cls = json.loads(data, object\_hook=lambda d: SimpleNamespace(\*\*d))

25 return cls

26

27 def insert(self):

28 while True:

29 id = random.randrange(0, 58 \*\* 7)

30 if not r.exists(id):

31 self.id = id

32 r.set(id, json.dumps(self.\_\_dict\_\_))

33 break

34

**Web server**

For simplicity, we use [Flask](https://flask.palletsprojects.com/en/2.0.x/quickstart/) as our web server.

1# \_\_init\_\_.py

2from flask import Flask, request, redirect, abort

3

4from .models import Link

5from .utils import BASE58

6

7app = Flask(\_\_name\_\_)

8BASE\_URL = 'http://localhost:5000/'

9

10@app.route("/api/link", methods=["POST"])

11def new\_link():

12 data = request.get\_json()

13 link = Link(data["longURL"])

14 link.insert()

15 short = BASE58.encode\_int(link.id)

16 data = {

17 'short': short,

18 'tinyurl': BASE\_URL + short

19 }

20 return {

21 'status': 'success',

22 'data': data

23 }

24

25@app.route("/<short>", methods=["GET"])

26def redirect\_link(short):

27 id = 0

28 try:

29 id = BASE58.decode\_int(short)

30 except KeyError:

31 abort(404)

32 link = Link.from\_redis(id)

33 if not link:

34 abort(404)

35 return redirect(link.longURL, code=302)

36

To run these code on your machine, you need to have python3 and pip installed. For Redis, you can use cloud services like [AWS](https://aws.amazon.com/redis/), [install](https://redis.io/topics/quickstart) on your machine, or use docker.

1# install python dependency

2pip install flask redis

3

4# run redis from docker

5docker run --name tiny-url-shortener-redis -p 6379:6379 -d redis

6

7# start web server

8FLASK\_APP=. flask run

9

You can clone the code on our [GitHub](https://github.com/realAlgoMonster/system-design/tree/main/tiny-url-shortener).

**Example**

Once the web server is started, we can use cURL to send a shorten request by

1curl --header "Content-Type: application/json" --request POST --data '{"longURL":"YOUR\_LONG\_URL"}' http://localhost:5000/api/link

and a successful response looks like

1{"data":{"short":"tuBQHmi","tinyurl":"http://localhost:5000/tuBQHmi"},"status":"success"}

Then you can use [http://localhost:5000/tuBQHmi](http://localhost:5000/tuBQHmi) to access the your long URL.

**Scalability**

Our web server is stateless. They simple process requests, read from the database and send the results back. There is no data persisted on the web servers. We can easily scale stateless web servers by adding more servers and putting them behind a load balancer.

For storage, we can scale Redis by using Redis Clusters, essentially splitting data into many redis instances as explained in the storage section above.