Designing a URL Shortner like Bit.ly

1.First Step: How to Convert Long URL → Short URL

Example

Long URL:

https://www.youtube.com/watch?v=abcd1234

Short URL:

bit.ly/nme

2.Given

- **Write operations per day = 20 million (20,000,000)**
- **Read : Write = 10 : 1**
- **1 record size = 100 bytes**
- **Time = 5 years**

STEP 1: Writes Per Second (QPS)

1 day = 86,400 seconds

Write QPS=20,000,00086,400\text{Write QPS} = \frac{20,000,000}{86,400}Write QPS=86,40020,000,000 ≈231 writes/sec≈ 231 \text{ writes/sec}≈231 writes/sec

STEP 2: Read Operations Per Day

Given ratio:

Read=10×WriteRead = 10 × WriteRead=10×Write Readperday=20,000,000×10=200,000,000Read per day = 20,000,000 × 10 = 200,000,000Readperday=20,000,000×10=200,000,000

STEP 3: Read QPS

ReadQPS=200,000,00086,400Read QPS = \frac{200,000,000}{86,400}ReadQPS=86,400200,000,000 ≈2315 reads/sec≈ 2315 \text{ reads/sec}≈2315 reads/sec

Type	QPS
Write	~231/sec
Read	~2315/sec

Peak traffic (×10):

- **Write peak ≈ 2,300/sec**
- **Read peak ≈ 23,000/sec**

STEP 4: Storage Calculation

Storage per record

100 bytes

Storage per day

20,000,000×100=2,000,000,000 bytes20,000,000 × 100 = 2,000,000,000 \text{ bytes}20,000,000×100=2,000,000,000 bytes =2GB/day= 2 GB/day=2GB/day

Storage per year

2GB×365=730GB/year2 GB × 365 = 730 GB/year2GB×365=730GB/year

Storage for 5 Years

730GB×5=3650GB730 GB × 5 = 3650 GB730GB×5=3650GB ≈3.65TB≈ 3.65 TB≈3.65TB

FINAL ANSWER

Traffic:

- **Write QPS ≈ 231/sec**
- **Read QPS ≈ 2315/sec**
- **Peak Read QPS ≈ 23K/sec**

Storage:

- **2 GB per day**
- **730 GB per year**
- **~3.6 TB for 5 years**

3.What is an API Endpoint?

An API endpoint is a URL where a client (browser/app) sends a request to the server.

Think of it like a door to your backend system.

In URL Shortener, We Need 2 Main APIs

1.POST API → Create Short URL

Purpose

User gives long URL, server returns short URL.

Endpoint

POST /shorten

Request Body

```
{
  "longUrl": "https://www.youtube.com/watch?v=abcd1234"
}
```

Response

```
{
  "shortUrl": "bit.ly/nme"
}
```

What Happens Internally?

1. **Generate unique ID**
2. **Convert to Base62**
3. **Store in DB**
4. **Return short URL**

2.GET API → Redirect to Long URL

Purpose

User opens short URL → server redirects to long URL.

Endpoint

GET /{shortCode}

Example:

GET /nme

How Redirect Works (Very Important)

Server Steps:

1. **Receive request:**

bit.ly/nme

1. **Look up DB:**

shortCode	longUrl
nme	https://youtube.com/...

1. **Send HTTP Redirect Response:**

HTTP 302 Redirect

Location: https://youtube.com/watch?v=abcd1234

Browser Action

Browser sees 302 and automatically opens the long URL.

Types of Redirect

301 Permanent Redirect

- **URL will never change**

302 Temporary Redirect (Used by bit.ly)

- **Allows analytics tracking**

URL shorteners use 302.

4.Goal

Convert:

https://www.youtube.com/watch?v=abcd1234

into

bit.ly/nme

There are 3 MAIN Ways to Generate Short URL

I'll explain the industry standard way first (this is what interviewers expect).

METHOD 1: Auto-Increment ID + Base62

STEP 1: Store Long URL and Get Unique ID (Deep Explanation)

What is Auto-Increment ID?

Auto-increment ID = Database automatically generates a unique number for every record.

Like:

Insert Order	Generated ID
First URL	1
Second URL	2
Third URL	3
...	...

Database Table Design

Table Name: urls

CREATE TABLE urls (

id BIGINT AUTO_INCREMENT PRIMARY KEY,

long_url TEXT NOT NULL

);

Meaning of Each Column

id BIGINT AUTO_INCREMENT

- **BIGINT → can store billions/trillions of values**
- **AUTO_INCREMENT → DB automatically increases number**

PRIMARY KEY

- **Ensures unique ID**
- **Fast lookup (indexing)**

long_url TEXT

- **Stores the original long URL**

What Happens When User Sends Long URL?

User sends:

https://www.youtube.com/watch?v=abcd1234

Step 1: Insert into Database

INSERT INTO urls(long_url)

VALUES ("https://www.youtube.com/watch?v=abcd1234");

What Database Does Internally?

Suppose table already has:

id	long_url
123	google.com
124	facebook.com

Now we insert new URL

DB automatically does:

id = 125

Final Table:

id	long_url
123	google.com
124	facebook.com
125	youtube.com

Why ID = 125 is Important?

This ID is:

- **Unique**
- **Small number**
- **Sequential**
- **Perfect to convert into short string**

Why Not Use Long URL Directly?

Long URL:

https://www.youtube.com/watch?v=abcd1234

Problems:

- **Very long**
- **Not user friendly**
- **Hard to share**

So we convert ID → short code

How DB Generates Auto Increment Internally?

Database maintains a counter:

current_id = 124

When insert happens:

current_id = current_id + 1

return 125

Concurrency Problem (Very Important)

What if 1 million users insert at same time?

DB uses:

- **Locks**
- **Atomic counters**
- **Transactions**

So:

User A → id 125

User B → id 126

User C → id 127

No duplicates.

Problem in Large Systems

Auto-increment in single DB does not scale.

Why?

- **Single point of failure**
- **High contention**

How Big Companies Solve It

1.Distributed ID Generators

Example:

- **Twitter Snowflake**
- **UUID**
- **Zookeeper**

DB Sharding with ID ranges

Server	ID Range
Server 1	1–1B
Server 2	1B–2B

Why We Need ID Before Base62?

Because:

Base62(ID) = ShortCode

No ID → No short code.

Full Step Flow

User → Server → DB

1. **User sends long URL**
2. **Server inserts into DB**
3. **DB generates unique ID**
4. **Server gets ID**
5. **Convert ID → Base62**
6. **Return short URL**

First, we store the long URL in the database. The database generates a unique auto-increment ID for each record. This ID acts as a unique identifier for the URL, and we later encode this ID using Base62 to generate a short URL.

Very Important

Why not store short URL directly?

Because:

- **Short URL depends on ID**
- **ID is guaranteed unique**
- **Avoid collisions**

Extra Point

Auto increment ID is predictable.
Attackers can guess:

bit.ly/1

bit.ly/2

bit.ly/3

So companies add:

- **Randomization**
- **Hashing layer**
- **Salt**

STEP 2: Convert ID to Base62

We convert 125 → "nme"

Base62 Characters

0-9 → 10 chars

a-z → 26 chars

A-Z → 26 chars

Total = 62 characters

Base62 Encoding Example

Convert 125 to Base62

125 % 62 = 1 → 'b'

125 / 62 = 2

2 % 62 = 2 → 'c'

2 / 62 = 0

Reverse result:

"cb"

Real systems produce something like: nme

STEP 3: Create Short URL

shortCode = base62(id)

shortUrl = "bit.ly/" + shortCode

Final:

bit.ly/nme

Why This Method is Best?

No collision
Short URLs
Fast
Scalable to billions

METHOD 2: Hashing (MD5 / SHA) to Generate Short URL

Instead of using a numeric ID, we convert the long URL into a hash string.

STEP 1: Take Long URL

User gives:

https://www.youtube.com/watch?v=abcd1234

STEP 2: Apply Hash Function (MD5 / SHA-1/256)

We run a cryptographic hash:

hash = MD5(longUrl)

What is a Hash Function?

A hash function:

- **Takes any input**
- **Produces fixed-size output**
- **Looks random**
- **Same input → same output**

Example MD5 Output

MD5("https://youtube.com/watch?v=abcd1234")

= e4d909c290d0fb1ca068ffaddf22cbd0

This is 32 hex characters (128-bit).

STEP 3: Take First 6 Characters

We shorten hash:

shortCode = e4d909

STEP 4: Create Short URL

shortUrl = bit.ly/e4d909

Full Flow

Long URL → Hash → Take first 6 chars → Short URL

Why Use Only 6 Characters?

Because:

Length	Possibilities
6 chars (Base62)	56 billion
7 chars	3.5 trillion

Small but still huge.

BIG PROBLEM 1: Collision

What is Collision?

Two different URLs produce same short code.

Example:

MD5(url1) → abcd12xxxx

MD5(url2) → abcd12yyyy

Both short codes become:

bit.ly/abcd12

❌ **WRONG mapping!**

Why Collision Happens?

Because:

- **Hash space is huge**
- **But we take only first 6 characters**
- **So probability increases**

Collision Example Table

Long URL	Hash	Short Code
google.com	abcd12xxx	abcd12
facebook.com	abcd12yyy	abcd12

Both conflict.

BIG PROBLEM 2: Retry Logic Needed

When collision happens:

Step 1:

Generate hash

Step 2:

Check DB if shortCode exists

Step 3:

If exists → regenerate (add salt, random value)

Example Retry Logic

hash = MD5(longURL + randomSalt)

BIG PROBLEM 3: Longer URLs

Hash output is longer than Base62 ID encoding.

BIG PROBLEM 4: Not Sequential

IDs:

1, 2, 3, 4...

Hash:

x9A2pL, Qw91Z, 7KjP0...

Hard to manage DB, no ordering.

BIG PROBLEM 5: Security Issues

Same long URL always generates same hash.

Anyone can guess URLs.

Why Companies DON’T Prefer Hashing?

Reason	Explanation
Collision	Same short code possible
Retry needed	Extra DB lookup
Performance	Hash computation
Not scalable	Hard to shard
Predictable	Same input → same output

Why Auto-Increment + Base62 is Better

Feature	Hashing	ID + Base62
Collision	Yes	No
Unique	Not guaranteed	Guaranteed
Performance	Slower	Fast
Scalability	Hard	Easy
Order	Random	Sequential

Another approach is to hash the long URL using MD5 or SHA and take the first few characters as the short code. However, this approach can cause collisions, so we need retry logic and additional checks. Because of these issues, most production systems prefer using an auto-increment ID with Base62 encoding.

Extra Point

Some companies use Hybrid Approach:

- Hash for deduplication
- ID for short code generation

METHOD 3: Random String (URL Shortener)

Idea

Instead of using auto-increment ID or hashing, we randomly generate a short code.

Example short codes:

aB9xQ2

ZkP1m7

9TqR0s

1.How Random Short Code is Generated

Step 1: Character Set

Same Base62:

a-z A-Z 0-9 → 62 characters

Step 2: Generate Random 6 Length String

Pseudo code:

```
string chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

string shortCode = "";


for(int i=0; i<6; i++){

   shortCode += chars[random(0,61)];

}
```

Step 3: Store in DB

short_code | long_url

aB9xQ2 | https://youtube.com/...

2.Capacity Calculation (Back of Envelope)

Length = 6
Characters = 62

Total = 62^6 = 56 Billion

Enough for internet scale.

3.But Big Problem: Collision

What is Collision?

Two different URLs get SAME random code.

Example:

youtube.com → aB9xQ2

google.com → aB9xQ2 ❌ (collision)

4.How Companies Handle Collision

Method A: Check DB and Retry

Algorithm:

generate random code

if code exists:

generate again

else:

save

Probability of Collision (Birthday Paradox)

When codes used = N
Total space = S

Collision probability increases when:

N ≈ sqrt(S)

For 6 chars:

sqrt(56B) ≈ 237,000

👉 **After ~200k URLs, collision risk starts increasing.**

5.Why Random String is NOT Preferred by Big Companies

Problem 1: Collision Handling Cost

DB lookup + retry = slow
At scale (millions per day) → huge overhead.

Problem 2: No Ordering

Auto-increment IDs are ordered
Random strings are chaotic → harder analytics.

Problem 3: Security Issues

Random generators must be cryptographically secure
Else attackers guess URLs.

Problem 4: Hotspot Problem

Random writes → no locality in DB → slower disks
Sequential IDs → faster disk writes (traditional DB principle)

6.When Random Method is Actually Used

Used when:

No centralized DB ID generator
Distributed systems without coordination
Serverless architectures
Temporary short URLs

Example:

- **CDN tokens**
- **Invite links**
- **Temporary share links**

7.Why Bitly DOES NOT Use Pure Random

Bitly uses:

Auto Increment ID
Base62 encoding

Because:

- **No collision**
- **Simple**
- **Predictable scaling**
- **DB friendly**

8.Hybrid Method (Industry Best Practice)

Generate random but guarantee uniqueness

Options:

UUID → Base62 encode

Snowflake ID → Base62

Random + DB uniqueness constraint

Random string method generates a short code using Base62 characters, but suffers from collision risk and scalability issues, so large-scale systems prefer auto-increment ID or Snowflake IDs with Base62 encoding.

Method	Collision	Scaling	Used in Big Companies
Auto Increment + Base62	No	★★★★★	YES
Hashing	Possible	★★	Rare
Random String	High	★★★	Rare at scale

Why SQL DB is best for a Short URL service (like bit.ly):

1. **Simple & fixed schema**
   Short URL data is very structured:
2. **short_code | long_url | created_at | expiry | user_id**

3. **SQL databases handle this kind of tabular, fixed structure *perfectly*.**
4. **Fast primary-key lookups**
   Short URL → Long URL is basically:
5. **SELECT long_url FROM urls WHERE short_code = 'abc123';**
6. **With primary key / index, this lookup is O(log n) (or near O(1) in practice).**
7. **Strong consistency required**
   When a user hits a short URL:
   - **It must redirect to the correct long URL**
   - **No stale or missing data allowed**

   SQL gives strong consistency + ACID guarantees, which is ideal here.
8. **Unique constraint is easy**
   Short codes must be unique.
9. **short_code UNIQUE**
10. **SQL handles uniqueness cleanly and safely.**
11. **Easy analytics & queries**
    You often need:
    - **count clicks**
    - **URLs created per day**
    - **expired links**

    SQL is great for aggregations & reporting.
12. **Low write, very high read pattern**
    - **Writes: only when URL is created**
    - **Reads: every redirect**

    SQL DBs scale very well for read-heavy workloads with indexes + replicas.
13. **Joins (if needed later)**
    If you add:
    - **users**
    - **plans**
    - **permissions**

    SQL makes relations and joins straightforward.

When SQL might NOT be enough

Problem at huge scale

Suppose your URL shortener gets billions of redirects per second.

If every redirect does:

SELECT long_url FROM urls WHERE short_code = 'abc123';

directly on SQL DB:

- **Database gets overloaded**
- **Disk reads increase**
- **Latency increases**
- **DB becomes bottleneck**

Even very strong SQL clusters cannot handle extreme read traffic efficiently.

Solution: Add Redis / Cache in front

Architecture becomes:

User → Load Balancer → Cache (Redis) → SQL DB

Flow

1. User clicks short URL abc123
2. System first checks Redis cache

GET abc123

Case 1: Cache HIT (most requests)

- Redis returns long URL instantly (microseconds)
- Redirect happens
- SQL DB not touched

Case 2: Cache MISS

- System queries SQL:
- SELECT long_url FROM urls WHERE short_code='abc123';
- Result stored in Redis:
- SET abc123 = long_url
- Redirect user

So next time it becomes cache hit.

Why Redis helps

1. RAM based → extremely fast
2. Handles millions of requests/sec
3. Removes read pressure from SQL
4. SQL becomes only source of truth
5. Cache stores hot URLs (frequently accessed ones)

Real-world analogy

Think:

- SQL DB = warehouse
- Redis = nearby shop

Customers (users) buy from shop (Redis).
Shop refills items from warehouse (SQL) when needed.