📘 Lecture 6: Designing a URL Shortener like Bit.ly

📌 Functional & Non-Functional Requirements

✅ Functional Requirements

- Accept **long URL** from user
    - Example:
      `www.example.com/api/v3/userid=1234user_id=harshal`
- Generate **short URL**
    - Example:
      `bit.ly/3xYzAb`
- Redirect **short URL** to original **long URL**

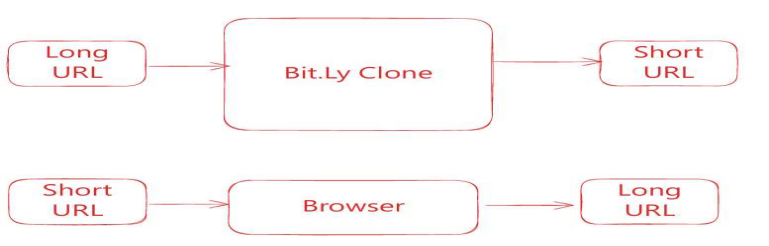
⚙️ Non-Functional Requirements

- Scalability (handle millions of URLs)
- High availability & reliability
- Low latency (fast redirects)

🧠 High-Level Design (HLD)

> 🔄 *Open-ended question → Discussion on requirements, functionality, and applications*

🔁 URL Redirection Flow :



`Long URL → Bit.ly (shortener) → Short URL`

`Short URL → Browser → Redirect to Long URL`

❓ Key Questions & Answers

Question	Answer
👥 How many URLs to target?	20 million URLs
🌐 Example of a long URL	www.example.com/api/v3/userid=1234user_id=harshal
📏 How short should the short URL be?	As short as possible
🔤 Allowed characters in short URL	0-9, a-z, A-Z

🧭 User Interaction Flow

1. 👤 User submits **long URL**
2. ⚙️ System generates **short URL**
3. 🌍 User enters **short URL** in browser
4. 🔁 System redirects to **long URL**

🛠️ Core Tasks

- 🔗 Generate short URL
- 🔁 Redirect short URL to long URL

Back of the Envelope Calculations – URL Shortener Design

⚡ 1. Write QPS (Queries Per Second)

📌 Given:

- **20 million URLs** are generated **per day**

📊 Calculation:

```
Write QPS = Total writes per day / Seconds per day

          = 20,000,000 / 86,400

          = (20 × 10⁶) / (86.4 × 10³)

          = 0.0002314815 × 10³

          ≈ 231.48
```

✅ Result:

Write QPS ≈ 232 writes/sec

🔍 2. Read QPS Calculation

📌 Given:

- **Read to Write ratio = 10:1**
  (i.e., every short URL is read ~10 times)

📊 Calculation:

```
Read QPS = Write QPS × 10

         = 232 × 10

         = 2,320 reads/sec
```

📝 Note: Every read results in a **database lookup**

✅ Result:

Read QPS ≈ 2,320 reads/sec

💾 3. Storage Estimation (For 5 Years)

📌 Given:

- 20 million records **per day**
- **5 years** of storage
- **1 record = 100 Bytes**

📊 Step 1: Total records in 5 years

```
Total Records = 20,000,000 × 365 × 5

              = 36,500,000,000 (36.5 billion records)
```

📊 Step 2: Total storage required

```
Total Storage = Total records × Size per record

              = 36.5 × 10⁹ × 100

              = 3.65 × 10¹² Bytes

              = 3.65 TB
```

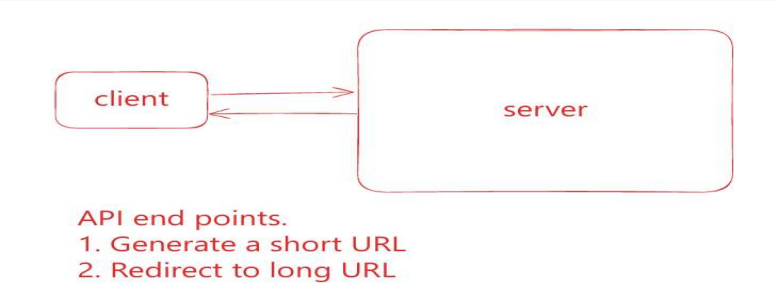
✅ Result:

Storage required ≈ 3.65 TB (for 5 years)

⚠️ Summary

Metric	Value
📝 **Write QPS**	≈ 232 writes/sec
📖 **Read QPS**	≈ 2,320 reads/sec
💽 **Storage (5 years)**	≈ 3.65 TB

Client ↔ Server Interaction & API Design



✅ Task

- Generate a **short URL**
- Redirect to the **long/original URL**

🧩 APIs Required

1. Generate a Short URL (POST Request)

🔶 **Endpoints Options:**

- `POST www.bit.ly/api/v3/generate?longURL=value` → (Query parameter)
- `POST www.bit.ly/api/v3/generate/{longURL}` → (Path variable)

- `POST www.bit.ly/api/v3/generate`
  Body: `{ "longURL": "value" }`

🔶 **How it works:**

Client sends:

```
{
  "longURL": "https://example.com/very/long/path"
}
```

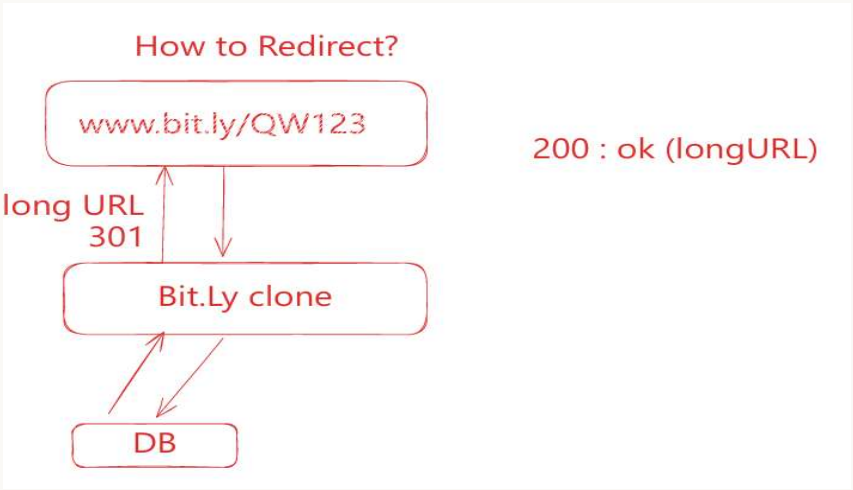
-

Server returns:

```
{
  "shortURL": "www.bit.ly/Qw23"
}
```

---

2. Redirect to Long URL (GET Request)



- Endpoint: `GET www.bit.ly/{shortURL}`
- Server response: Redirect to the original long URL

🔀 How Redirection Works

- **Client** can't redirect using `200 OK` — that only means success, not redirection
- To redirect, **server must use HTTP status codes** understood by browsers
- Server hits the **database** to get the long URL from the short URL

🧭 Redirection Status Codes

🔶 301 – Permanent Redirect

- URL: `www.bit.ly/Qw23`
- Server responds with **301** status code
- Tells browser: this short URL will **always** go to the same long URL
- Browser does **client-side caching** (like cookies or local storage)
- On next visit: browser doesn't ask server — it directly redirects

---

🔶 302 – Temporary Redirect

- URL: `www.bit.ly/Qw23`
- Server responds with **302** status code
- Tells browser: this redirection is **temporary**
- Browser **does not cache** the result
- Each time user uses the short URL:
    - It hits the server again
    - Server can send updated or different long URL if needed

---

⚠️ Important Notes

- Use **301** when redirection is fixed & won't change
- Use **302** when redirection might change in future

- **302 is useful** when you want to:
    - Track every hit
    - Change long URL later
    - Avoid client-side caching

⚖️ 301 vs 302 Redirect – Kab Kaunsa Use Karein?

🔴 301 – Permanent Redirect

✅ **Flow:**

```
User → bit.ly/Qw23 → (Browser handles redirect) → Long URL
```

✅ **Pros:**

- ✅ Browser **cache** karta hai → **No repeated server calls**
- ✅ **Less load** on server/database
- ✅ **Fast** redirection for users

❌ **Cons:**

- ❌ **No tracking** (server ko hit hi nahi hota)
- ❌ **Cannot update** long URL later

🟡 302 – Temporary Redirect

✅ **Flow:**

```
User → bit.ly/Qw23 → (Server handles redirect) → Long URL
```

✅ **Pros:**

- ✅ **Track** every request (for analytics)
- ✅ **Change** long URL in future if needed
- ✅ Useful for **marketing**, **A/B testing**, **user behavior tracking**

❌ **Cons:**

- ❌ Browser doesn't cache → **Every visit hits server**
- ❌ Needs **scalable backend**

✅ Recommendation: Use `302` Redirect

> Because it gives **flexibility**, **metrics**, and **future control** over URLs.

🛠️ System Design: Initial Flow Breakdown

🔷 Task 1: URL Shortening

🧭 **Flow:**

```
Long URL → [Service] → Short URL
```

📩 **API:**

POST `www.bit.ly/api/v3/generate`

📥 **Request Body:**

```
{
  "longURL": "https://example.com/very/long/path"
}
```

📤 **Response:**

```
{
  "shortURL": "www.bit.ly/Qw23"
}
```

🧠 How to Store Mapping Internally

- Use **HashMap** or **HashTable**

```
Key   → shortURL
Value → longURL
```

📌 **Example:**

```
task.get("Qw23") → "https://example.com/very/long/path"
```

🔷 Task 2: Redirect to Long URL

🧭 Flow:

`Short URL → [Service] → Redirect to Long URL`

🌐 API:

GET `www.bit.ly/{shortURL}`

⚙️ Backend Steps:

1. Extract `{shortURL}` from request
2. Lookup `shortURL` in HashMap
3. Fetch corresponding `longURL`
4. Respond with **HTTP 302 Redirect**

📤 Response Example:

```
HTTP/1.1 302 Found
Location: https://example.com/very/long/path
```

🧠 Important Concepts to Remember

Feature	301 Redirect	302 Redirect
Redirection Type	Permanent	Temporary
Can Track Users?	❌ No	✅ Yes
Can Update URL?	❌ No	✅ Yes
Browser Caching	✅ Yes	❌ No
Use Case	SEO, Static URLs	Analytics, A/B testing, Dynamic URLs

How to Generate a Short URL from a Long URL

🔷 Concept: Hash Function-Based URL Shortening

🧮 Process:

```
Long URL → Hash Function → Short URL
```

Example:

```
Input:  https://example.com/very/long/path
Hash:   e.g. MD5, SHA-1, SHA-256
Output: bit.ly/a1B2c
```

🗂️ HashMap Storage Strategy

🔑 Key (Short URL)	📍 Value (Long URL)
bit.ly/a1B2c	https://example.com/very/long/path

✅ This mapping is stored in the **database**

🛠️ URL Shortening Phase (Storing the URL)

🔄 Flow:

```
[User sends Long URL]
         ↓
[Apply Hash Function → Get Short URL]
         ↓
[Store in HashMap / DB: short → long]
         ↓
[Return Short URL to user]
```

📌 Technologies:

- Use **cryptographic hash functions**:
  `MD5`, `SHA-1`, `SHA-256` (to ensure uniqueness)

🚀 URL Redirection Phase (Reading from DB)

🔄 Flow:

```
[User hits Short URL]
        ↓
[Lookup: map.get(shortURL)]
        ↓
[Fetch Long URL]
        ↓
[Redirect user to Long URL]
```

⚠️ Problem: In-Memory HashMap Is Not Scalable

❌ Limitations:

- Stored in **RAM**
- Works only **on a single server**
- As data grows (millions/billions of URLs), memory will overflow
- Cannot **scale horizontally**

🛢️ Scalable Solution: Use a NoSQL Database

✅ Use:

- **Amazon DynamoDB** (Key-Value store) — recommended
- **MongoDB** (Document store) — more complex

> 🔄 Replace in-memory HashMap with **persistent, scalable DB**

🧠 High-Level Design (HLD): Distributed System

📦 Architecture Components:

- Multiple **Web Servers** (for high availability)
- Multiple **Database Servers**
- **DB Replication** (for fault tolerance)

- **Sharding** (to distribute data)

🧩 Problem in Sharding: Which DB Stores Which URL?

✅ Use: Consistent Hashing

🔁 Now We Need Two Hash Functions:

Purpose	Hash Function	Role
🔍 **To find which DB shard to store in**	`SHA-1`	Distribute data across DBs
🔐 **To generate the short URL**	`MD5`	Create a unique short key

---

🔄 Full Flow:

```
[Long URL]
     ↓
1. Apply SHA-1 → Identify DB shard
2. Apply MD5   → Generate short URL
     ↓
[Store shortURL → longURL in selected DB]
```

✅ This ensures:

- Efficient **load balancing**
- Consistent **short URL generation**
- Horizontally **scalable architecture**

✅ Summary

Step	Action	Tool
1️⃣	Short URL Generation	MD5 / SHA-256
2️⃣	Sharding / DB Selection	SHA-1 + Consistent Hashing
3️⃣	Storage	NoSQL DB (e.g., DynamoDB)

4	Redirect Lookup	shortURL → longURL mapping
5	Scaling	Distributed Servers + DB Replication

Better Design – NoSQL → SQL DB

🔁 Why Move from NoSQL to SQL?

```
Client → Server → Short URL
```

🧩 Step 1: Short URL Generation (Initial Logic – NoSQL)

```
(1) Long URL → Hash Function → Short URL
```

DB Used: **DynamoDB (NoSQL)**

Short URL	Long URL
short	long

⚠️ Problem:

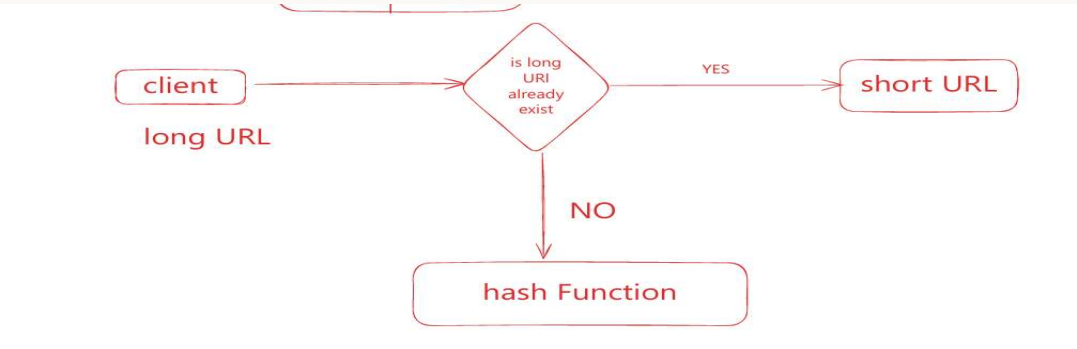
- अगर user **same Long URL** दोबारा भेजे
- और **POST request** करे कि "मुझे फिर से short URL दो"
- तो:

```
Long URL → Same hash function → Same output
```

- लेकिन system logic फिर से **hash function apply** करेगा
- और **same या दूसरा short URL** बना देगा
- जो पहले ही generate किया जा चुका है

🔁 Same hash function हमेशा **same number** generate करेगा
लेकिन अगर check नहीं किया गया कि **long URL already है**, तो **duplicate entry** बन सकती है

✅ Solution:



Improved Flow:

```
Client → Long URL → Check: Already present?

If YES → Return existing Short URL
If NO  → Apply hash function → Generate Short URL → Store in DB
```

- इससे बार-बार same Long URL पर **duplicate short URLs** नहीं बनते
- DB में पहले से मौजूद Short URL को ही **reuse** किया जाता है

⚠️ Problem in Earlier NoSQL Logic:

- पुरानी mapping:

```
Key   → Short URL
Value → Long URL
```

- लेकिन अगर आपको check करना है कि **Long URL पहले से है या नहीं**, तो:

आपको **Long URL पर query** करनी होगी
जो कि **NoSQL में मुश्किल** है या inefficient हो सकती है

🏭 Production-Level Application Requirements:

(1) Query:

For a particular input (user), how many URLs have been generated till now?

✅ SQL supports such **complex queries** efficiently

---

(2) Performance:

Write operations are faster in SQL compared to NoSQL in this scenario

---

🧰 SQL Table Design (With Indexing)

Column Name	Description
ID	Primary Key (Auto ID)
Short URL	Encoded (shortened) URL
Long URL	Original full URL

- Corresponding ID system के लिए indexing useful होगी

🤔 Hash Functions – Long URL → Short URL

Hashing Algorithms:

- `MD5`, `SHA-1`, `SHA-256`

🔶 Problem:

Hash function se jo value milti hai, wo bahut **badi hoti hai**, lekin humein **short URL chahiye hota hai.**

🔣 Allowed Characters in Short URL:

```
(0-9) → 10
(a-z) → 26
(A-Z) → 26
Total = 62 characters
```

❓ Interview Insight: Keep Short URL “as small as possible”

Example:
`www.bit.ly/asd` (3 characters)

🔢 Permutations: How Many URLs Can Be Formed?

Characters (n)	Possibilities (62^n)
1 char	62
2 chars	3,844
3 chars	238,328
4 chars	14.7 million
5 chars	916 million
6 chars	**~56.8 billion**

> Requirement: Store **36.5 billion URLs** → So, take **n = 6**

✅ Format:

`www.bit.ly/xxxxxx → 6-character Short URL`

> Agar `n < 6`, toh system required number of records store nahi kar paayega.

⚙️ Hash Function Output Trimming

Problem:

Hash value → Long (e.g. `ASFWS34233PGWPD3432PQLRS`)
But we only need **6 characters**

✅ Solution:

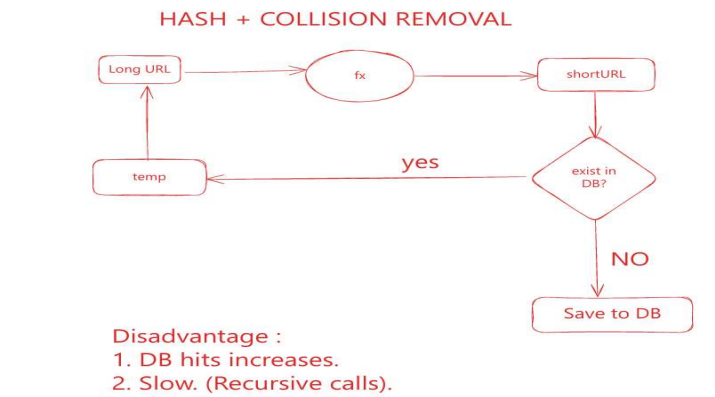
Trim first 6 characters
Example:

`Long Hash → ASFWS3`

⚠️ Collision Possibility:

Same 6-character hash value doosri URL ke liye bhi aa sakti hai.

🛠️ Solution: Hashing + Collision Removal



🔁 Flow:

```
Long URL
   ↓
Hash Function → 6-char Short URL
   ↓
Check in DB:
    ├─ Yes → Return existing Short URL
    └─ No  → Save to DB
          ↓
       If collision:
       - Append temp string (e.g. random or incremental)
       - Re-hash
       - Repeat check
```

🔁 Recursive Hashing:

```
Try 1: Hash("www.example.com/page") → ASFWS3 → Exists? → Yes
Try 2: Hash("www.example.com/page1") → RTH9KQ → Exists? → No → Save
```

⚠️ Disadvantages:

- 🔁 **More DB Hits**: Har collision par DB ko check karna padta hai.

- 🐢 **Slower Performance**: Recursive hashing se time badh jaata hai.

🔢 BASE-X (Base-62) Approach

Characters Allowed:

```
Total Characters = 62
Digits: 0–9 (10)
Lowercase: a–z (26)
Uppercase: A–Z (26)
```

🔄 Base-10 → Base-62 Conversion:

Decimal	Base-62
0–9	0–9
10–35	a–z
36–61	A–Z

🔧 Example: Convert `14920` into Base-62

Step-by-Step Division:

```
14920 ÷ 62 = 240, remainder = 60 → Z
240 ÷ 62   = 3,   remainder = 54 → S
3 ÷ 62     = 0,   remainder = 3  → 3
```

✅ Final Base-62 Result:

```
Read from bottom → top → 3SZ
```

📎 Mapping in SQL Table:

ID	Short URL	Long URL
14920	3SZ	www.example.com

🔹 **Short URL:** www.bit.ly/3SZ
🔹 **Long URL:** www.example.com

✅ Advantages of Base-62 Approach

1. **No Collisions**
   (Each ID is unique → Unique Short URL)

❌ Disadvantages

1. Requires a **Unique ID Generator**
2. **Short URL length is not fixed** (as number increases, length increases)
3. **Predictable**: Next URL can be guessed (e.g. 3SZ → 3T0)

   ✅ Solution: Use **Random Unique ID** instead of incremental

---

🧪 Example (Random Unique ID):

```
Long URL: http://www.example.com/get/code/custID=124
Generated Unique ID: 14920
Short URL = base62(14920) = 3SZ
```

⚙️ SQL Table Structure:

ID	Short URL	Long URL
14920	3SZ	www.example.com

🏗️ Distributed System Issue

When multiple DBs (shards) are used:

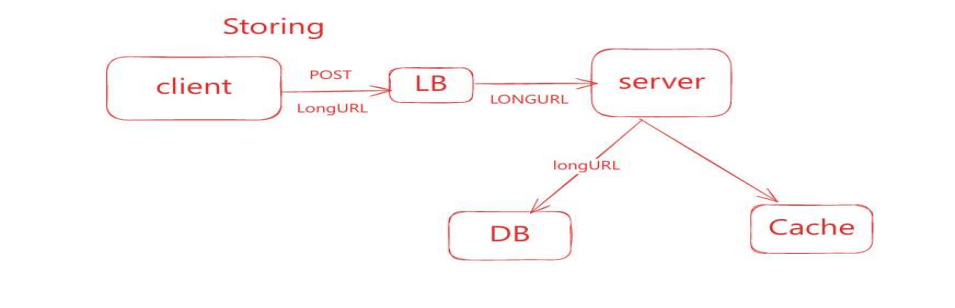
- They may generate the **same IDs independently**
- Can cause **Short URL duplication**

  ✅ Solution: Use a **Centralized ID Generator**

🖼️ High-Level System Architecture (HLD)

📤 Storing (POST Flow)

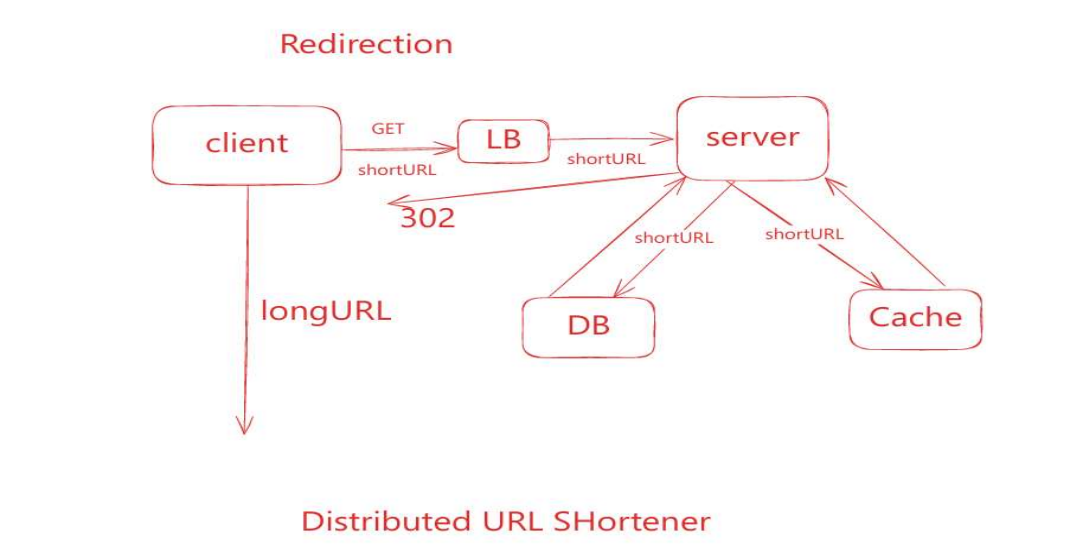
```
Client → POST → Load Balancer → Server
          ↓                        ↓
       Long URL                → Logging
                                   ↓
                               DB + Cache
```



📥 Redirection (GET Flow)

```
Client → GET (Short URL) → LB → Server
                              ↓
                        Find Long URL
                              ↓
                    Response: 302 Redirect
                              ↓
                     Client goes to Long URL
```

302: Temporary redirect → useful for tracking hits



🌐 Final Design: Distributed URL Shortener

System supports:

- Centralized ID generation
- Load balancing
- Caching
- Database sharding (with caution)

❓ Common Interview Question

Q: What if Unique ID generates a value that gives more than 6 characters in Base-62?

A: Not possible.

```
62⁶ = 56.8 Billion unique URLs possible
→ We only need to store ~36.5 Billion URLs
→ 6 characters in base-62 are enough
```

Quick Recap :

✅ Requirements

📌 Functional:

- Accept Long URL
- Generate Short URL (e.g., bit.ly/3xYzAb)
- Redirect Short URL to Long URL

⚙️ Non-Functional:

- **Scalable** (Millions of URLs/day)
- **High Availability**
- **Low Latency** (Fast redirects)

🧠 Key Concepts

Metric	Value
Write QPS	~232/sec
Read QPS	~2,320/sec
Storage (5 years)	~3.65 TB
Read:Write Ratio	10:1
Characters Allowed	0–9, a–z, A–Z = 62 total
Short URL Length	6 characters ($62^6 \approx$ 56B)

🛠️ URL Generation Methods

🔷 1. Hash Function Approach

- Use: MD5 / SHA-1 / SHA-256 → Get hash → Trim first 6 chars
- 🔁 Handle Collisions by:
    - Appending temp strings
    - Re-hashing until unique
- ❌ Disadvantage: More DB hits, slower due to recursion

🔷 2. Base-62 Encoding (ID → Short URL)

- Convert auto-increment ID to base-62 (e.g., 14920 → 3SZ)
- ✅ Collision-free
- ❌ Predictable, variable-length, needs central ID generator

🧩 API Design

🔶 POST `/generate`

Request:

```
{ "longURL": "https://example.com/very/long/path" }
```

Response:

```
{ "shortURL": "bit.ly/3xYzAb" }
```

🔸 GET `/{shortURL}`

→ Redirects to original long URL

🔁 Redirection: 301 vs 302

Type	Cache?	Trackable?	Use When
301	✅ Yes	❌ No	Permanent redirect (SEO)
302	❌ No	✅ Yes	Analytics, A/B testing

✅ Recommended: 302 → Flexible + Trackable

🏗️ Scalable Architecture (HLD)

1. **Client → Load Balancer → Servers**
2. **Centralized ID Generator**
3. **DB + Caching Layer (e.g., Redis)**
4. **Sharded DB + Consistent Hashing for Long URL storage**
5. **Use NoSQL (DynamoDB) or SQL (MySQL/PostgreSQL) as needed**

💡 Optimization Notes

- Use **HashMap in-memory** only for prototype
- Use **NoSQL** for fast writes, **SQL** for analytics/queries
- Handle same long URL with **deduplication logic**

- Use **Base62** for compact, user-friendly URLs
- 6-char short URL supports **up to 56.8 Billion** entries