System Design Interview Guide (Detailed)

Walkthrough: Design a URL Shortener (Detailed)

1. Requirements Gathering

Functional Requirements:

- Shorten a long URL to a compact version.
- Redirect a short URL to its corresponding original URL.
- Optionally track metadata such as number of clicks, timestamp, browser info, etc.

Non-Functional Requirements:

- High availability and low latency.
- System should scale to handle millions of URLs and redirections.
- Persistence and durability of mappings.

2. High-Level Architecture

Client -> Load Balancer -> Application Server

[Redis Cache]

[Primary Database (SQL/NoSQL)]

3. Components Breakdown

a) API Layer:

- POST /shorten

  Request: {"long_url": "https://openai.com/blog"}

Response: {"short_url": "https://short.ly/abc123"}

- GET /abc123

  Response: HTTP 301 redirect to original long URL

b) Short Code Generation:

- Option 1: Use an auto-incrementing counter and convert it to base62.
- Option 2: Use a hash function (MD5/SHA256) with collision handling.
- Option 3: UUIDs with base62 encoding.

c) Encoding:

- Base62 encoding uses [0-9][a-z][A-Z] => allows ~56B unique values with just 6 characters.

d) Storage:

- SQL Table (for reliability):

```
  CREATE TABLE short_urls (
    id SERIAL PRIMARY KEY,
    short_code VARCHAR(10) UNIQUE,
    long_url TEXT,
    created_at TIMESTAMP
  );
```

- NoSQL Alternative (e.g. DynamoDB):

  Partition key: short_code, Value: long_url

e) Caching:

- Use Redis to store frequently accessed mappings for fast retrieval.

4. Flow

a) URL Shortening:

- Client sends POST with long URL.
- Backend generates unique short code.
- Save mapping in database.
- Store in cache.
- Return short URL.

b) Redirection:

- Client requests short URL.
- Backend checks Redis cache:
  - If hit: redirect.
  - If miss: query DB, update cache, then redirect.

5. Scaling and Reliability

- Use load balancers to distribute traffic.
- Horizontally scale app servers.
- Use replication and sharding in database.
- Use distributed cache with TTL.
- Implement retries and fallback mechanisms.

6. Security and Abuse Prevention

- Validate URLs.
- Set expiration TTL on short links.
- Rate-limit abusive clients.

- Add reporting endpoints for spam detection.

7. Analytics (Optional)

- Async logging of access metadata: IP, timestamp, user-agent.

- Store in separate DB or logging pipeline (Kafka + ELK).

System Design Interview Cheat Sheet (Detailed)

System Design Interview Cheat Sheet

A. Interview Flow

1. Clarify Requirements

- Ask: What is the goal of the system?

- Clarify: Traffic load, latency, data volume, consistency needs.

2. Define APIs & Use Cases

- Define main endpoints (REST or gRPC).

- List core use cases and failure cases.

3. High-Level Design

- Draw boxes: Client -> Load Balancer -> App Server -> DB/Cache

- Call out major components (API Gateway, DB, Caching, Queue, Worker, CDN)

4. Deep Dive Components

- How does data flow?

- How to handle scale and failures?

- What are the tradeoffs?

5. Bottlenecks & Tradeoffs

- Where can it fail? How to recover?
- Consistency vs Availability (CAP theorem)
- Latency vs Throughput tradeoffs

6. Wrap-Up

- Recap your design
- Suggest improvements
- Show metrics to evaluate success

B. Key Concepts & Tools

- Load Balancer: NGINX, HAProxy, AWS ELB
- Cache: Redis, Memcached
- Database: PostgreSQL, MySQL, MongoDB, DynamoDB
- Message Queue: Kafka, RabbitMQ, SQS
- Object Store: S3, GCS
- CDN: Cloudflare, Akamai
- Rate Limiting: Token bucket, Leaky bucket
- Authentication: JWT, OAuth2, API key
- Monitoring: Prometheus, Grafana, ELK stack

C. Interview Tips

- Always state assumptions clearly.
- Use tradeoffs language: This increases write speed but may delay consistency.

- Think out loud, iterate and adapt based on feedback.

- Finish with: If I had more time, Id consider X...