**URL Shortening Service - High-Level System Design**

### **Introduction**

A URL shortening service (like bit.ly) converts long URLs into compact, easy-to-share links. Users submit a long URL and receive a shortened version. When the short URL is visited, the service redirects to the original URL. This document presents the high-level design for such a system.

### **Components**

1. **Front-End**
   * Simple web UI for users to input long URLs and retrieve shortened versions.
   * Displays analytics (click count, referrers, etc.).
2. **Load Balancer**
   * Distributes incoming traffic across multiple API servers for high availability.
3. **Back-End**
   * API server handling URL creation and redirection.
   * Generates unique short keys using a hybrid of counter-based + hashing/random string.
   * Manages user authentication (optional).
4. **Database**
   * Stores mappings between short keys and long URLs.
   * Stores metadata (creation timestamp, click count, expiration).
   * Uses master-slave replication for high availability.
5. **Cache**
   * In-memory cache (e.g., Redis) to speed up redirects for frequently accessed short URLs.
6. **Analytics Service**
   * Tracks click events, geo-location, device type.
   * Eventual consistency (no need for strict real-time updates).
7. **CDN**
   * Serves static content and common redirects, reducing latency for global users.
8. **Monitoring & Logging**
   * Tracks performance, usage trends, and system errors.
9. **Cleanup Service**
   * Periodically removes expired or inactive URLs.

### **API Endpoints**

1. **POST /create**

| { "long\_url": "https://www.example.com" } |
| --- |

**Request**:

**Response**:

| { "short\_url": "https://short.ly/abc123" } |
| --- |

1. **GET /:short\_key**
   * Redirects to the original URL.

Returns HTTP 301/302 redirect:  
  
 HTTP/1.1 301 Moved Permanently

Location: https://www.example.com

### **Database Schema**

| CREATE TABLE urls (  id SERIAL PRIMARY KEY,  long\_url TEXT NOT NULL,  short\_key VARCHAR(10) UNIQUE NOT NULL,  created\_at TIMESTAMP DEFAULT NOW(),  click\_count INT DEFAULT 0,  expires\_at TIMESTAMP NULL ); |
| --- |

### **Key Generation Techniques**

* **Counter-based**: Auto-incrementing counter encoded in base62.
* **Random string**: 6-10 character random alphanumeric.
* **Hashing**: Hash of the long URL (with salt).
* **Hybrid**: Combines counter-based and random for uniqueness + unpredictability.

### 

### 

### 

### **Scalability Considerations**

* **Load Balancing**: Nginx/HAProxy distributes traffic.
* **Horizontal Scaling**: Multiple API servers.
* **Caching**: Redis for popular URLs.
* **Database Sharding**: Partitioned based on short\_key hash.
* **Replication**: Master-slave DB setup for HA.
* **CDN**: Accelerates content delivery.
* **Async Analytics**: Eventual consistency via Kafka queues.

### **Edge Cases & Security**

* **Duplicate URLs**: Return existing short URL.
* **Malicious Inputs**:  
  + URL validation
  + Block unsafe domains
  + CAPTCHA/rate limiting for bots.
* **Redirect Loops**: Prevent nesting of short URLs.
* **Abuse**: Rate limiting per IP.

### **Assumptions & Trade-offs**

* **Assumptions**: High read-heavy system; 10-100x more GETs than POSTs.
* **Trade-offs**:  
  + Real-time vs. eventual analytics (for scalability).
  + URL expiration policies reduce storage but may impact UX.

### **Conclusion**

A URL shortening service must be simple, fast, secure, and globally scalable. Caching and replication allow the system to handle millions of daily requests. CDN and load balancers ensure low latency. Addressing edge cases and using smart key generation ensures reliability and security.

**Architecture Diagram** (conceptual):

| Clients --> Load Balancer --> API Servers --> [ Redis Cache ]  --> [ Master DB <-> Slave DB ]  --> [ Analytics Service / Kafka ]  --> [ Cleanup Service ]  --> [ CDN (Static Content) ] |
| --- |

This architecture ensures low latency, high availability, and fault tolerance.