Mastering System Design

Design a URL Shortener (aka TinyURL)

Introduction to TinyURL

- TinyURL is a URL shortening service that converts long URLs into short, unique links for easy sharing and tracking.
- Example:
  - https://www.example.com/very/long/path/to/resource → https://tinyurl.com/abc123
- Why it's Useful:
  - Compact Links: Perfect for platforms like Twitter and SMS
  - Improved UX: Easy to share, clean, and readable
  - Click Tracking: Analytics for link performance
  - Branding: Custom domains for enterprises
  - Cross-Channel Friendly: Works across emails, chats, and print
- How it Works:
  - User submits a long URL
  - System generates a unique short key
  - Stores mapping in database
  - Redirects short URL to original

Functional Requirements for TinyURL

- **Shorten a URL**: Accept a valid long URL and return a shortened URL.
  - Example: Input: https://example.com/article?id=1234 → Output: https://tinyurl.com/xYz12
- **Redirect to Original URL**: When accessing the short URL, redirect to the original long URL.
- **Prevent Duplicate Short URLs**: If the same long URL is submitted, return the same short URL or handle according to configuration (unless custom alias is used).
- **User Authentication**: Allow users to register/login to manage URLs, view analytics, and set expiration.

Non-Functional Requirements for TinyURL

- **High Availability**: System must be available 24/7 with >99.9% uptime.
- **Performance & Low Latency**: URL redirection should occur in milliseconds. Shortening URLs should be near-instantaneous.
- **Scalability**: System must handle millions or billions of URLs, supporting high read volume (redirects) and moderate write volume (URL shortening).
- **Reliability**: Ensure data persistence and no data loss even during failures using durable storage and backups.

Unique URL Generation Strategies

- Random String Generation: Creates a fixed-length string from random characters
  - ✅ Unpredictable, no obvious pattern
  - ⚠️ Risk of collisions, requires collision handling
  - 🔍 Okay for unpredictability, but adds complexity
- UUID (Universally Unique Identifier): 128-bit globally unique identifier (e.g., 123e4567-e89b-12d3-a456...)
  - ✅ Guaranteed uniqueness, no central coordination
  - ⚠️ Very long, not user-friendly
  - 🔍 Not ideal for TinyURL due to length
- Hashing with Salt: Hashes the original URL (e.g., SHA-256 + salt)
  - ✅ Unique, secure, hard to reverse
  - ⚠️ May not be short, collision possible, needs mapping storage
  - 🔍 Useful for security-focused cases, but not optimal for shortening
- Base62 Encoding: Converts incrementing ID to Base62 (0-9, a-z, A-Z)
  - ✅ Short, compact, deterministic, easy to implement
  - ⚠️ Needs counter management to avoid collisions
  - 🔍 **Recommended for TinyURL (fast, scalable)**

Estimating Scale & Identifying Bottlenecks

- 👥 Estimated User Traffic
  - Daily Active Users (DAU): ~10 million
  - Monthly Active Users (MAU): ~300 million
  - New Short URLs/day: ~100,000 (1% of DAU)
  - Redirect Requests/day: ~50 million (5 per user)
- 🧠 Memory Requirement (Hot URL Cache)
  - Cache top 1M most accessed URLs
  - Each mapping ≈ 500 bytes
  - Total memory ≈ 500 MB
- 🌐 Network Bandwidth (Redirects)
  - 50M redirects/day × 700 bytes = ~35 GB/day
  - Avg throughput: ~0.4 MB/sec
  - Peak throughput: ~5 MB/sec
- 💾 Storage Requirement (URL Mapping DB)
  - 100K new URLs/day × 500 bytes = ~50 MB/day
  - Yearly data ≈ 18 GB raw + overhead
  - Round up to ~50 GB/year (with indexes, logs, backups)

Bottleneck Identification

- High read volume → Focus on cache and fast DB reads
- Write throughput is moderate → Ensure consistency
- Latency sensitivity in redirects → Low-latency infra needed
- Plan for burst traffic with autoscaling & CDN support

API Design – Create Short URL

- Accepts a long URL and returns a shortened URL.
- Endpoint: **POST /api/shorten**
- Request (JSON):

```
{
  "long_url": "https://www.example.com/article?id=123",
}
```

- Response (JSON):

```
{
  "short_url": "https://tinyurl.com/my-alias"
}
```

API Design – Redirect

- Redirect to Original URL
- Endpoint: **GET /:short_key**
  - Example: GET /abc123
- Behavior:
  - Looks up the original long URL using the short key.
  - Returns a 302 HTTP Redirect to the long URL.

API Design – Delete

- Delete a Short URL (Optional)
- Endpoint: **DELETE /api/url/:short_key**
- Behavior:
  - Deletes the mapping if user is authenticated and owns the URL.
  - Requires Bearer token for authorization.

User Authentication APIs

- User Registration
- Endpoint: **POST /api/auth/register**

```
Request:
{
  "name": "John Doe",
  "email": "user@example.com",
  "password": "securePassword123"
}
Response:
{
  "message": "Registration successful"
}
```

- User Login
- Endpoint: **POST /api/auth/login**

```
Request:
{
  "email": "user@example.com",
  "password": "securePassword123"
}
Response:
{
  "access_token": "<JWT>",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

- Secure Endpoints with Bearer Token:

```
Authorization: Bearer <access_token>
```

High-Level System Design – Overview

- API Gateway: Entry point for all clients; handles routing, rate limiting, auth
- URL Shortener Service: Contains logic for key generation, duplicate checking, alias validation
- Redirect Service: High-performance resolver for short keys → long URLs
- Database: Persistent store for all mappings, users, metadata
- Cache Layer: Redis/Memcached for top N frequently accessed URLs
- Auth Service: Manages user login, JWT tokens, and sessions

Collision Handling in Distributed URL Generation - Hello Zookeeper

- Why Collisions Happen
  - Multiple service instances generating IDs independently → risk of duplicates
  - No global coordination → Base62 encoding same ID → incorrect URL mapping
- What is Zookeeper?
  - Distributed coordination service by Apache
  - Ensures synchronization across nodes in a distributed system
- Zookeeper as a Solution
  - 🌐 Atomic ID generation using Zookeeper-managed global counter
  - ✅ Guarantees each instance gets a unique ID
  - 🧩 Uses znodes to store and manage counters
  - 🔐 Supports distributed locking to serialize ID generation
- Flow
  - Service requests next ID from Zookeeper
  - Zookeeper increments global counter atomically
  - ID is Base62 encoded and used as TinyURL
  - (Optional) Mapping stored in DB

Strategic Tech & Infra Decisions for TinyURL

- Database:
  - SQL (e.g., PostgreSQL with auto-increment IDs)
  - NoSQL (e.g., Redis for caching, DynamoDB for scalability)
  - Cache: Redis or Memcached for high-speed lookup
- Scalability & Performance
  - Horizontal scaling for URL generation services
- High Availability
  - Load Balancer to distribute traffic across service instances
  - Replication in DB to avoid single points of failure
  - Failover-ready infrastructure using cloud-managed DBs or services

The Final Design - TinyURL

- URL Generation Flow
  - 👤 User submits long URL via API
  - 🔐 API Gateway handles authentication, throttling
  - 🏗️ URL Generation Service:
    - Requests a unique ID from Zookeeper (ensures no collisions)
    - Encodes it (e.g., Base62) to create the short URL
    - Stores the mapping in the Database (Short URL → Long URL)
  - 📦 Response sent back with the generated TinyURL
- 📥 Redirection Flow
  - 👤 User hits a short URL (e.g., tinyurl.com/abc123)
  - 🔐 API Gateway forwards to Redirection Service
  - ⚡ Redirection Service:
    - Checks Cache for the short URL (fast-path)
    - If not found, queries the Database (cold-path)
  - 🚀 Redirects user to the original long URL (HTTP 302)