**URL SHORTENER**

**Generate Short URLs**: Convert long URLs to short ones using a hashing method, ensuring each short URL is unique.

**Retrieve Long URLs:** When a user requests a short URL, the system fetches the corresponding long URL from storage.

Core Components:

**Hashing for Uniqueness:** Use a hashing algorithm to create short URLs. To avoid collisions (where different URLs generate the same hash), the system must check if a generated hash already exists.

**Database for Mappings:** Store the mappings between long URLs and their corresponding short URLs in a relational database. The database structure typically includes an auto-incrementing ID for efficient access.

**Caching with Redis:** Implement caching to store recently accessed URL mappings, reducing database load and speeding up response times for frequently used short URLs.

**Redirect Strategies:** Use HTTP response codes (301 for permanent redirects, 302 for temporary redirects) to handle URL redirection. A 301 is more cache-efficient, while a 302 is useful for tracking purposes.

**Scalability and Performance Considerations:**

**High Volume Optimization:** Adjust the character length of short URLs to handle a large number of mappings as the service grows.

**Collision Handling:** Consider appending strings or modifying the hash algorithm to reduce the likelihood of collisions.

**Database Capacity:** As traffic increases, improve the database’s ability to handle a larger volume of reads and writes.

**User Experience Enhancements**:

**Caching Benefits:** Since users often access recently generated URLs, caching can significantly enhance the speed and efficiency of the service.

**Future Directions:** Expand the system design to include more complex features, such as user notifications or analytics.

**Designing a URL Shortening Service**

A URL shortening service transforms long URLs into shorter, more manageable links while maintaining the ability to redirect to the original URL. The service must meet both functional and non-functional requirements to ensure high performance, scalability, and user satisfaction.

**Functional Requirements**

Shorten Long URLs: The service should generate a short, unique URL for any given long URL.

Redirect to Original URLs: When a user accesses the short URL, the service should redirect them to the corresponding long URL efficiently and reliably.

### ****Non-Functional Requirements****

* **High Availability:** The service must be available 24/7 with minimal downtime to handle millions of requests per day.
* **Low Latency:** URL redirection should occur within milliseconds to provide a seamless user experience.
* **Scalability:** The service should support a growing number of users and URLs without performance degradation.

**Designing a URL Shortening Service:**

**URL Length and Token Generation**

* **Length:** A 7-character URL supports up to 3.5 trillion unique URLs.
* **Token Generation:** Use a Token Service with Base62 encoding or hashing (e.g., MD5, SHA-256) to ensure uniqueness and prevent collisions.

**Database Management**

* **Database Choice:** Use **Cassandra** for scalability and high availability, or **MySQL with sharding** for horizontal distribution.
* **Schema:** Include fields for Short URL (Primary Key), Long URL, Creation Timestamp, and optional Access Count for analytics.

**Performance Optimization**

* **Caching:** Utilize **Redis** or **Memcached** to cache frequently accessed URLs and reduce database load.
* **Batch Processing:** Aggregate analytics data asynchronously in batches to minimize I/O operations.

**Analytics Integration**

* Implement asynchronous tracking for user behavior (click rates, geography) using **Apache Kafka** for real-time data or **Apache Spark** for batch processing.

**Redirect Handling**

* **HTTP Status Codes:** Use **301** for permanent redirects and **302** for temporary redirects or tracking.

**Designing a Chat Application:**

**Core Features**

* **Private and Group Chats:** Support for direct messaging and group conversations.
* **User Presence Tracking:** Real-time status updates (online/offline) for users.

**Capacity Estimation**

* Anticipate 150 GB of data daily from 5 million users, each sending an average of 80 messages.

**Database Management**

* **Choice of Database:** Use a NoSQL database like **Cassandra** for high write performance and scalability, as the system involves more writes than reads.

**System Architecture**

* **Real-Time Messaging:** Utilize **WebSockets** for instant message delivery.
* **User Mapping and Presence Detection:** Dedicated services manage user sessions and detect presence.
* **Notification System:** Sends alerts to offline users through a specialized service.

**Scalability and Performance**

* Design the system to scale horizontally to accommodate growing user numbers.
* Focus on minimizing latency for real-time interactions, even at the cost of some consistency.

**Real-Time Messaging:** Balance minimal latency with robust architecture to enhance user experience.

**API Design:** Ensure well-structured APIs for messaging, group management, and user presence to enable smooth client-server communication.

**Database Selection:** Opt for a NoSQL database like Cassandra to handle high write loads with minimal overhead.

**Designing WhatsApp**

**Core Features**

* **Group Messaging:** Supports groups of up to 200 users for seamless group communication.
* **Multimedia Sharing:** Enables sending of images and videos, enhancing user engagement.
* **Read Receipts:** Notifies users when their messages are delivered and read.
* **Last Seen Status:** Shows user availability, adding context to communication.
* **Temporary Chats:** Offers options for temporary message storage, emphasizing user privacy.

**System Architecture**

* **WebSockets:** Utilizes WebSockets for real-time messaging to ensure fast and reliable message delivery.
* **Microservices Architecture:** Implements a microservices approach, breaking down the application into specialized services (e.g., session management, group services) to improve scalability, flexibility, and maintainability.

**Performance and Scalability**

* **Load Balancing:** Distributes user connections efficiently across servers to manage millions of users.
* **Decoupled Services:** Separates functionalities into distinct microservices to enhance the system’s resilience and ease of maintenance.
* **Message Queues:** Uses message queues to ensure reliable message delivery, especially during peak traffic times, and handles retries in case of failures.

**User Activity Tracking**

* **Last Seen Feature:** Tracks user activity to provide real-time information about user presence, balancing the need for privacy with communication transparency.

**Security and Reliability**

* **Security Protocols:** Implements strong security measures at the gateway level to protect user data against unauthorized access.
* **Message Prioritization:** During high-load periods, prioritizes essential messages to maintain system performance and user satisfaction.

**One-to-One Messaging:** Supports direct messaging between users, ensuring private communication.

**Read Receipts:** Provides users with feedback on whether their messages have been delivered and read.

**Media Sharing:** Allows users to send images, videos, and other media files, enhancing interaction.

**Push Notifications:** Sends real-time alerts to users, such as message delivery notifications and user activity updates.

**System Architecture**

* **Client-Server Model:** Utilizes a client-server architecture where messages are routed through a central server. This model supports efficient message delivery, storage, and retrieval.
* **Message Flow:** Detailed sequence diagrams are used to map the process of message delivery, from the sender's initiation to the recipient's confirmation. For example, when User A sends a message to User B, the server stores the message temporarily and confirms delivery once User B is online.
* **Transient Storage for Offline Users:** Messages are stored temporarily on the server for users who are offline. Once the recipient comes online, the stored messages are delivered immediately.

**Temporary Storage:** Media files like images and videos are stored temporarily on the server. When a user is offline, URLs for media files are forwarded to them upon reconnection, ensuring efficient use of storage space.

**FIFO Queue:** A First-In-First-Out (FIFO) queue is used to manage the order of media messages, ensuring that messages are delivered in the order they were sent.

**Server Requirements:** Plans for scaling server infrastructure to handle millions of concurrent connections, considering factors such as load balancing, fault tolerance, and distributed architecture.

**Server Failures:** Designs systems to handle server failures gracefully. For instance, if a server goes down, clients are programmed to reconnect automatically, maintaining a seamless user experience.

**Transient Storage and Message Delivery:** Using transient storage ensures messages reach offline users reliably. For instance, if User A sends a message to User B while they are offline, the server stores the message until User B comes online, ensuring it isn’t lost.

**Scalability Analysis:** Performing a scalability analysis, like estimating server needs based on traffic projections, ensures that the system can handle high user concurrency. For instance, planning for 10,000 concurrent users per server helps in deciding the number of servers required.