System Design Day 1: Scalability Basics

1. **What is Scalability?**

- **Definition**: The ability of a system to **handle increasing load** (traffic, data, users) by **adding resources**.
- **Goal**: Maintain performance as usage grows.

2.
Types of Scalability

Type	Description	Example
Vertical Scaling (Scale Up)	Add more power (CPU, RAM) to a single server	Upgrading EC2 instance from t3.medium to m5.4xlarge
Horizontal Scaling (Scale Out)	Add more machines to distribute load	Adding more app servers behind a load balancer. Also adding replicas which we are using in Kubernetes (SOAM) or adding multiple nodes(AGMS, QBEP)

Horizontal scaling is generally preferred for
distributed systems and cloud-native apps.

3. **Scalability Dimensions**

- **Traffic Scalability**: Can the system handle more concurrent users?
- **Data Scalability**: Can it store and process more data?
- **Feature Scalability**: Can it accommodate new features without major rewrites?

4. Design Strategies to Achieve Scalability

1. **Load Balancing**
    - Distributes traffic across multiple servers.
    - Tools: NGINX, HAProxy, AWS ELB.
2. **Caching**
    - Reduces load on backend services.
    - Tools: Apache Ignite, Redis, Memcached, CDN for static content.
3. **Database Sharding**
    - Split large databases into smaller, manageable parts.
    - Based on user ID, region, etc.
4. **Asynchronous Processing**
    - Offload long-running tasks via message queues.
    - Tools: Kafka, RabbitMQ, AWS SQS.
5. **Stateless Services**
    - Avoid storing session data on the app server.
    - Makes horizontal scaling easy.
6. **Auto-scaling**
    - Automatically scale resources based on metrics.
    - AWS Auto Scaling, Kubernetes HPA.

5. Metrics to Monitor Scalability

- **Latency**
- **Throughput**
- **CPU/Memory usage**
- **Queue length / request rate**
- **Error rate under load**

Best Example Graphana Dashboards

6. **Scalability Challenges**

- Database bottlenecks (single write-master)
- Stateful services
- Poorly designed APIs (chatty, synchronous)
- Network bandwidth limitations

---

7. **Scalability Examples in Real Life**

- **Netflix** uses microservices and CDNs to stream to millions.
- **Amazon** shards product and inventory data by region.
- **Instagram** scaled to millions using horizontal scaling + Redis + sharded PostgreSQL.

8. **Explain load balancing in simple terms**

Load balancing is a technique used to distribute incoming work or traffic across multiple servers to ensure no single server gets overwhelmed, improving performance, reliability, and scalability.

Simple Explanation

Imagine a busy restaurant with one waiter trying to serve all customers—things would get slow and chaotic. Now, if the restaurant hires multiple waiters and a host assigns customers evenly to them, service becomes faster and smoother. Load balancing works similarly in computing:

- **What it does**: It spreads requests (e.g., website visits, API calls) across multiple servers or resources.
- **Why it's needed**: To handle high traffic, prevent server crashes, and reduce response times.
- **How it works**: A **load balancer** (like a traffic cop) sits between users and servers, directing each request to an available server based on rules (e.g., which server is least busy).

Key Concepts in Simple Terms

1. **Traffic Distribution**: When users access your Java/Spring Boot app, the load balancer sends their requests to different servers to avoid overloading one.
   - Example: If 100 users visit your e-commerce API, the load balancer might send 25 to Server A, 25 to Server B, and so on.
2. **Types of Load Balancers**:
   - **Hardware**: Physical devices (e.g., F5 load balancers).
   - **Software**: Programs like Nginx, HAProxy, or cloud-based (AWS Elastic Load Balancer).
3. **Balancing Methods**:
   - **Round Robin**: Sends requests to servers one by one in order (e.g., Server 1, Server 2, Server 3, repeat).
   - **Least Connections**: Sends requests to the server with the fewest active users.
   - **IP Hash**: Sends a user to the same server based on their IP for consistency.
4. **Benefits**:
   - **Speed**: Distributing work reduces wait times.
   - **Reliability**: If one server fails, the load balancer reroutes traffic to others.
   - **Scalability**: Add more servers to handle more users without downtime.

Example in Your Context (Java Developer)

In a Java/Spring Boot microservices app, you might have multiple instances of your service running (e.g., on AWS EC2). A load balancer (like AWS ELB) directs user requests to these instances:

- User visits your app → ELB checks which instance is least busy → Sends request to that instance → User gets a response.
- If one instance crashes, ELB routes traffic to other instances, keeping your app online.

Real-World Analogy

Think of a supermarket with multiple checkout counters. A load balancer is like an employee directing customers to the shortest line, ensuring no counter gets overwhelmed and everyone gets served quickly.

Why It Matters for System Design

As a Senior Java Developer transitioning to system design, load balancing is critical for designing scalable systems. In interviews, you'll discuss how to use it to handle high traffic (e.g., in a URL shortener or e-commerce platform). You can relate it to your experience with Spring Boot APIs deployed in a clustered environment or behind a tool like Nginx.

If you want a deeper dive (e.g., how AWS ELB works with Spring Boot) or a diagram of load balancing in a system, let me know!