System Design Day 10: Interview Prep

🎯 Purpose of System Design Interviews

- Evaluate your ability to:
  1. **Translate vague problem statements** into clear technical requirements.
  2. **Design scalable, efficient architectures** to solve those problems.
  3. **Communicate and defend your design choices** effectively.

💼 Why They Matter

- Critical for **senior engineering roles**.
- Reflects your readiness to **design real-world, large-scale systems** (e.g., Instagram, Uber, Twitter).
- Poor performance can lead to **lower level offers** even with experience.

✅ Preparation Strategy

1. 🛠️ Practice

- **Design real-world systems** on paper: Instagram, Gmail, etc.
- Sketch system components: clients, app servers, caching layers, databases, CDNs, etc.
- Walk through **data flows** (e.g., image uploads, news feed sorting).

2. 📚 Master Common System Design Patterns

- Load balancing, database sharding, CDN usage, caching, etc.
- Understand **trade-offs**, benefits, and drawbacks of each pattern.
- Prepare to **defend technical choices** (e.g., Redis vs. Memcached).

3. 📐 Use Visual Tools Effectively

- Be fluent with whiteboards, diagramming tools (like Miro, Excalidraw).
- Practice sketching systems **clearly and quickly**.

4. 🎭 Do Mock Interviews

- Simulate real interview conditions.
- Get feedback on:
  - Technical design
  - Time management
  - Communication skills
  - Ability to respond to curveballs

🧠 During the Interview: Key Behaviors

👂 Ask Clarifying Questions

- Understand use cases, constraints, and priorities.
- Don’t jump straight to the solution.

⏱️ Manage Time Wisely

- Be aware of time sinks.
- Keep design process moving forward.

🧩 Think Aloud

- Verbalize trade-offs, assumptions, and reasoning.
- Help the interviewer **follow your logic** step-by-step.

🧱 Structure Your Design

- Break into components (clients, APIs, databases, queues, etc.).
- Show data flow and bottlenecks.
- Stay at a **high level**, avoid low-level implementation.

Clarify Scale First – Summary:

Before jumping into a system design solution, always **clarify the scale of the system** you're expected to design. This sets the foundation for all your architectural decisions.

🔍 Why It Matters:

- **Scale determines complexity** – A system handling 1K users vs. 100M users needs very different designs.
- Helps decide on:
  - **Data storage models** (SQL vs NoSQL)
  - **Caching layers**
  - **Load balancing**
  - **Replication, sharding, CDN**
- Avoids **overengineering** or **underengineering** your solution.

💬 What to Ask:

- **Number of users** (daily active, concurrent)
- **Read/write ratio** (e.g., 90% reads?)
- **Expected QPS** (queries per second)
- **Latency requirements**
- **Data size & growth** (e.g., GBs vs petabytes)

✅ When to Do It:

- **Right after understanding the problem statement**
- Before discussing architecture or drawing diagrams

🎯 Outcome:

Clarifying scale first ensures your design is **realistic, efficient, and tailored** to the problem. It also shows the interviewer that you're thinking **like a real engineer** who builds for production.

Discussing CAP Trade-offs – Summary

When designing distributed systems, it's crucial to understand and explain the **CAP Theorem** trade-offs:

⚖️ What is CAP?

CAP stands for:

- **C**onsistency – Every read gets the most recent write.
- **A**vailability – Every request receives a response (success or failure).
- **P**artition Tolerance – The system continues to function even if parts of the network are unreachable.

> In a distributed system, you can only fully guarantee **two of the three at any time.**

🔁 Trade-off Scenarios:

1. CP (Consistency + Partition Tolerance)

- Prioritizes data correctness over availability.
- **Use case:** Financial systems (e.g., banking, payments).
- **Trade-off:** System may reject requests during a partition to maintain data accuracy.

2. AP (Availability + Partition Tolerance)

- Prioritizes uptime and response even with stale data.
- **Use case:** Social media feeds, product listings.
- **Trade-off:** Data may be eventually consistent across nodes.

3. CA (Consistency + Availability)

- Only possible in **non-distributed** or **partition-free** systems.
- Not realistic in modern internet-scale systems.

🧠 How to Discuss in Interviews:

- Identify which trade-offs make sense for the **use case**.

- Justify your choice based on:
  - **User expectations**
  - **Business priorities**
  - **Latency tolerance**
- Show awareness of **eventual consistency**, **read-repair**, **quorum-based systems**, etc.

✅ Takeaway:

Discussing CAP trade-offs shows your ability to design **resilient, realistic distributed systems** that meet business needs under failure conditions.

✅ How to Explain Spring-Based Designs in System Design Interviews

When asked to describe your architecture or design using **Spring Framework (especially Spring Boot)** in interviews, the key is to show how Spring enables **modular, scalable, maintainable**, and **production-ready** applications.

🧱 1. Start with the Layered Architecture

Explain how Spring promotes clean separation of concerns:

- **Controller Layer** – Handles HTTP requests (e.g., `@RestController`)
- **Service Layer** – Business logic (e.g., `@Service`)
- **Repository/DAO Layer** – Data access using Spring Data JPA or JDBC (e.g., `@Repository`)
- **Model Layer** – Domain entities (e.g., `@Entity`)

This modular design makes your system **testable**, **scalable**, and **easy to evolve**.

⚙️ 2. Key Spring Features to Highlight

🔁 Dependency Injection

- Achieved via `@Autowired`, constructor injection.

- Enables **loose coupling**, making code easier to test and swap implementations.

🚀 Spring Boot Auto-Configuration

- Reduces boilerplate and speeds up development.
- Automatically configures beans, DB connections, embedded servers (e.g., Tomcat), etc.

💾 Spring Data (JPA, Mongo, etc.)

- Simplifies data persistence.
- Repository pattern with **auto-generated queries** (`findByEmail()`, etc.)
- Supports relational and NoSQL databases.

🧰 Spring Security

- Easily integrate authentication & authorization.
- Useful for securing REST APIs using **JWT**, OAuth2, etc.

🔀 Spring Cloud (for Microservices)

- Show you can build **distributed systems** with:
  - **Service Discovery** (Eureka)
  - **API Gateway** (Spring Cloud Gateway)
  - **Config Server**
  - **Circuit Breaker** (Resilience4j)
  - **Distributed Tracing**

🌐 3. Use Spring in Your System Design Components

When discussing components like:

- **API Gateway** → Use *Spring Cloud Gateway*
- **Service Layer** → Use *Spring Boot REST APIs*
- **Data Layer** → Use *Spring Data JPA* or *MongoDB*
- **Asynchronous Processing** → Use *Spring Events*, *@Async*, or *Spring + Kafka*

- **Scheduling Jobs** → Use *@Scheduled*
- **Security** → Use *Spring Security + JWT*

💬 4. How to Communicate in the Interview

Structure your answer like this:

> "I'd use Spring Boot to build modular RESTful services. The system follows a layered architecture: the controller handles incoming requests, the service layer encapsulates business logic, and the repository uses Spring Data JPA for persistence. For microservices, I'd use Spring Cloud for service discovery, centralized config, and gateway routing. Security is handled via Spring Security with JWT-based auth."

🧠 Bonus Tips

- Mention **profiles** (`application-dev.yml`, `application-prod.yml`) for environment-specific configs.
- Highlight **Actuator endpoints** for health checks and monitoring.
- Show understanding of **transaction management** (`@Transactional`) especially in payment or critical flows.
- If using Kafka or RabbitMQ, describe **Spring Integration or Spring Cloud Stream**.

How to Justify **Sharding** in a System Design Interview

Sharding is a **horizontal partitioning** technique where large datasets are split across multiple databases (or nodes) to improve **scalability**, **performance**, and **availability**.

🎯 When to Justify Sharding

You should propose sharding when:

- The **dataset is huge** (e.g., billions of rows).
- A single database node can't handle the **read/write load**.
- **Query latency increases** due to too much data on one machine.
- You're hitting **storage or compute limits** of a single DB server.
- You want to **scale writes horizontally** (e.g., user-generated content systems like Twitter, Instagram, YouTube).

📌 How to Justify It in an Interview

Structure your answer using these talking points:

🔍 1. Problem Statement

> "As our user base and data volume grow, a single database becomes a bottleneck for both performance and storage."

🚀 2. Sharding as a Solution

> "To scale horizontally, I would shard the data across multiple database instances. This reduces the load per server and allows us to serve more users efficiently."

📊 3. Benefits of Sharding

- **Improved performance** – Queries are faster because each shard holds less data.
- **Write scalability** – Writes can be distributed across shards.
- **High availability** – Failure of one shard doesn't bring the whole system down.
- **Better resource utilization** – CPU, memory, and storage per shard are optimized.

🧠 4. **Shard Key Strategy**

> "I'd carefully choose a shard key (e.g., user_id) to ensure even distribution and avoid hot spots."

- **Good shard keys**: user ID, tenant ID
- **Bad shard keys**: timestamps, status (can lead to skewed data)

⚠️ 5. **Trade-offs and Challenges**

A good answer should show awareness of limitations:

- **Cross-shard joins** are difficult.
- **Rebalancing shards** can be complex.
- **Choosing the right shard key** is critical for avoiding uneven load.

🧵 6. **Real Example**

> “In a social media system like Instagram, posts can be sharded by user_id. That way, a user’s posts are stored in one shard, making profile page queries fast and independent of other users' data.”

✅ Final Summary to Say

> “Sharding helps us overcome the limits of a single-node database by enabling horizontal scaling. With a good shard key and proper planning, it drastically improves performance and supports long-term system growth.”
