Tips

Tuesday, 1 October 2024 11:28 AM

Mind of interviewer:

- Not technical expertise
- How we approach problems in general
- How we work with others like soft skills
- If they tell something which is not you suggested then ack and accept that don't defend and argue

Key things to keep in mind:

- Don't ask for time and sit idle for 5 mins, speak up and tell what's in your mind
- Start by clarifying requirement (ask lot of questions)
  - Start with repeating the question and telling your understanding of it.. Like youtube, google search
  - Break it down into specific requirements
  - Who are the customers? Are they all around the world? What time we can expect peak traffic? What are the use cases that we need to do?
- Scalability
  - How much video we are expecting
  - What will be the traffic expected
- Latency
- Availability
- Budget
- While talking about technology, work backward
  - When approaching system design "working backwards," you start by thinking from the **end-user's interface** and work toward the backend components like **CDNs**, **distributed data systems**, and the systems that populate those data stores. This methodology helps ensure that the design decisions are made with the end-user experience in mind.
- Toolchest with cloud solutions no need to build from scratch

When to go for what technology?(Scaling / transaction rate / latency / availability)

- If scaling is limited then a single DB
  - If scaling is massive we need horizontally scaling solution, some distributed system
  - If truly massive then we need to go for cloud storage system like S3, who is managing the scaling
- If transaction rate is high, what is the volume of users
  - If large number of transactions then we need large number of servers at the front end
  - Have to see how its distributed across servers and across the world
  - Internal tools doesn't need such complexity
- Latency - how quickly you need to return the values
  - This tells the need for caching and CDN usage
  - If there should not be any buffering then CDN is needed, cache will help on saving money in actual equipment
- Availability show how much down time you can tolerate
  - Higher the availability then it need to be stored redundant in multiple racks, multiple servers, multiple locations to avoid availability issues
  - If the master is down then should the slave should take the position of master automatically..

Aspect	**Vertical Scaling (Scaling Up)**	**Horizontal Scaling (Scaling Out)**
Method	Increase resources (CPU, RAM) of a single server	Add more servers or machines to share the load
Complexity	Simple to implement	More complex (requires load balancing, distributed systems)
Limit	Limited by hardware capacity	No theoretical limit (can keep adding servers)
Downtime	Possible downtime during upgrades	No downtime if done properly
Fault Tolerance	Single point of failure	High fault tolerance (redundancy across servers)
Use Case	Smaller systems, monolithic applications	Large-scale, distributed, cloud-native applications

Factor	**Definition**	**Effect on System Design**	**Priority in Large-Scale Systems**	**Priority in Small-Scale Systems**

			Scale Systems	Scale Systems
Scalability	The ability to handle increased loads by adding resources	Systems need to support growth without performance degradation. Horizontal scaling is crucial for large systems, while vertical scaling may suffice for small ones.	Horizontal scaling, microservices, cloud-native architectures	Vertical scaling, simpler architectures, fewer nodes
Transactions	The volume and complexity of operations processed by the system	High transaction volumes need databases with strong consistency models, whereas low-volume systems can prioritize simplicity.	Distributed databases, eventual consistency, NoSQL databases	Single-server databases, strong consistency, simpler transactions
Latency	The time taken to process requests	Minimizing response time is crucial, especially in large systems. Latency affects user experience and performance, requiring load balancing and caching.	Low-latency systems, use of caching (e.g., Redis), load balancers, CDNs	Acceptable latency can be higher, fewer optimizations needed
Availability	Ensuring the system remains operational and accessible	High availability (HA) is critical for large systems, requiring redundancy, failover mechanisms, and distributed systems. Smaller systems can tolerate more downtime.	High availability, fault-tolerant, multi-region deployments	Moderate availability, simpler failover solutions, may tolerate downtime

- **Large-scale systems** prioritize **horizontal scaling**, **high availability**, **low latency**, and the ability to handle high **transaction** volumes.
- **Small-scale systems** focus on **vertical scaling**, simpler architectures, and can tolerate higher latency and downtime.

Storing and stream vedio's:

- DB to store meta data - horizontally distributed db eg., BigTable(NoSQL DB)
- Omit buffering while streaming vedio - using CDN which be storing near the location
- Storing billions of vedio's can be done using cloud storage
- Vedio's have many resolution and many screen size so each vedio should be encoded into different ways and should be duplicated to edge locations around world on CDN's
- Alternate for CDN's as it would be costly is unpopular videos can be hosted in own servers in their own data centers
- Raw Vedio's are uploaded in the temp data store(some cloud storage) and then enqueued into a message queue(Amazon SQS) for transcoding the vedio from queue one by one and keeps in distrubed cloud storage

Cache

- Memcache - In-memory key-value store used to cache frequently accessed data in RAM.
  - Memcache is ideal for applications that need to access the same data frequently, but don't require strong consistency.
  - It's commonly used for session management, caching database query results, and reducing latency in web applications.

In-memory data processing

- Apache spark - Apache Spark is a distributed data processing engine designed for large-scale data processing. It provides in-memory computation, making it faster than traditional Hadoop MapReduce for many workloads. It supports a wide range of tasks, such as batch processing, streaming, machine learning, and graph processing.
  - Spark is used for big data processing when you need high performance (through in-memory computation) and scalability.

NoSQL DB with low latency

- dynamo db - DynamoDB is a fully managed NoSQL database service by AWS. It offers low-latency, scalable, and highly available key-value and document store. It automatically handles replication, scaling, and backup, and it's designed for applications requiring consistent, fast performance.
  - DynamoDB is used in applications where high throughput and scalability are required. It's suitable for use cases like session management, real-time bidding, e-commerce transactions. It is favored for its automatic scaling, high availability, and serverless nature.

Data storage large scale

- amazon s3 data lake - Amazon S3 (Simple Storage Service) is an object storage service, and a data lake built on S3 is used to store and manage large volumes of raw or processed data. A data lake on S3 can hold structured, semi-structured

to store and manage large volumes of raw or processed data. A data lake on S3 can hold structured, semi-structured, and unstructured data, and it integrates with other AWS services for analytics, machine learning, and processing

- Amazon S3 is used for creating data lakes in scenarios where you need to store vast amounts of data at low cost and with high durability. It's used in data-intensive applications like big data analytics, archiving, backup, and machine learning

Message QUEUE

- Amazon SQS - Queue - Amazon Simple Queue Service (SQS) is a fully managed message queuing service that allows applications to communicate by sending and receiving messages through a queue. It is designed to decouple the components of distributed systems, making it easier to build and scale microservices, serverless applications, and distributed systems.

NoSQL	Relational DB
Store data in a flexible format, such as key-value pairs, documents, wide-column stores, or graph structures. Data structures can vary within the same collection (no fixed schema).	Store data in structured, predefined tables
Often scale horizontally (distributing data across multiple servers).	Typically scale vertically (adding more CPU, memory, etc. to a single server)
Schemaless or flexible schema, Prioritize availability and partition tolerance over strict consistency (BASE model). Some NoSQL databases may offer eventual consistency.	Support ACID (Atomicity, Consistency, Isolation, Durability) transactions, which guarantee data reliability and consistency.
Do not use JOINs. Relationships are often handled at the application level or by embedding related data.	Utilize foreign keys and JOINs to manage relationships between tables.
Store as document, json.. { "_id": 1, "first_name": "John", "last_name": "Doe", "email": "john.doe@example.com", "address": "123 Main St, Cityville", "phone_number": "555-1234"}	
Query : db.customers.find({ _id: 1 });	
More filters in query : db.customers.find({ _id: 1 }, { first_name: 1, last_name: 1, orders: 1 });	SELECT customers.first_name, customers.last_name, orders.order_id, orders.order_date FROM customers JOIN orders ON customers.customer_id = orders.customer_id

End-User Interface (Frontend)

This is the layer the user interacts with. It can be a web application, mobile app, or desktop app.

Key Technologies:

- **Web Applications**:
  - **React.js**, **Angular.js**, **Vue.js**
  - **Pros**: High-performance, large ecosystem, good developer tools.
  - **Cons**: Complexity in maintaining state, large apps may suffer performance issues.
- **Mobile Applications**:
  - **React Native**, **Flutter**, **Swift/Objective-C (iOS)**, **Kotlin/Java (Android)**
  - **Pros**: Cross-platform (React Native/Flutter), good performance (native iOS/Android apps).
  - **Cons**: Limited in terms of platform-specific features (React Native/Flutter), higher maintenance for native apps.

Considerations:

- **Responsiveness**: The frontend should be optimized for different screen sizes and devices.
- **Performance**: Ensure minimal latency and fast load times by leveraging techniques like code splitting, lazy loading, etc.

Content Delivery Network (CDN)

A CDN caches and serves static resources (e.g., images, JavaScript, CSS) to users from servers close to their geographical location, reducing latency and improving load times.

Popular CDN Providers:

- **Cloudflare**, **AWS CloudFront**, **Akamai**, **Fastly**

Pros:

- **Reduced Latency**: Content is cached closer to the user, improving load speeds.
- **Scalability**: CDNs can handle large volumes of traffic, reducing load on the origin servers.
- **DDoS Protection**: Many CDNs offer built-in security features like DDoS protection.

Cons:

- **Cache Invalidation**: Propagating content changes can sometimes take time across distributed CDN nodes.
- **Cost**: For very large systems, CDN costs can become substantial depending on the traffic volume.

Distributed Data Systems (Databases)

Distributed databases handle data replication across multiple regions and servers, providing high availability and fault tolerance.

Popular Distributed Databases:

- **Cassandra**: A NoSQL database known for high write throughput and scalability.
  - **Pros**: Highly scalable, decentralized, fault-tolerant, no single point of failure.
  - **Cons**: Complex data modeling, eventual consistency, high write latency at scale.
- **CockroachDB**: A cloud-native SQL database that offers horizontal scalability.
  - **Pros**: Supports SQL, highly available, strong consistency, cloud-native.
  - **Cons**: Higher complexity, newer in terms of ecosystem maturity.
- **Amazon DynamoDB**: Fully managed NoSQL service on AWS.
  - **Pros**: Fully managed, seamless scalability, low latency at scale.
  - **Cons**: Expensive for large workloads, limited query capabilities compared to relational DBs.
- **Google Bigtable**: A NoSQL database service suited for high-throughput workloads.
  - **Pros**: Scalable, designed for large-scale workloads, low latency.
  - **Cons**: High complexity, not ideal for transactional or relational workloads.

Pros of Distributed Databases:

- **Fault Tolerance**: Data is replicated across multiple nodes, providing high availability.
- **Horizontal Scalability**: Adding more nodes to the cluster increases capacity.

Cons:

- **Eventual Consistency**: Many distributed databases trade off strong consistency for availability, meaning data changes might not be immediately reflected across all nodes.
- **Complexity**: Managing distributed systems can be complex, especially when dealing with network partitioning, consistency, and replication.

Systems to Populate the Data Store

These are the systems responsible for ingesting, processing, and writing data into the distributed database.

Data Pipelines:

- **Apache Kafka**: A distributed streaming platform used for building real-time data pipelines.
  - **Pros**: High throughput, fault-tolerant, scalable.
  - **Cons**: Requires complex management, higher learning curve for teams unfamiliar with it.
- **Apache Flink / Apache Spark**: Stream and batch processing frameworks used for ETL jobs.
  - **Pros**: Handles real-time and batch processing, distributed, fault-tolerant.
  - **Cons**: Higher resource consumption, complex configuration and management.
- **AWS Lambda**: Serverless compute service that runs code in response to events, ideal for data processing.
  - **Pros**: Fully managed, auto-scaling, pay-per-use.
  - **Cons**: Stateless, can be expensive at scale, limited execution time (max 15 minutes).

Pros:

- **High Throughput**: These systems are designed for processing large amounts of data in real-time or near real-time.
- **Fault Tolerance**: Many data pipeline systems (like Kafka, Spark) are designed to handle failures and recover automatically.

Cons:

- **Resource Intensive**: Large-scale data pipelines can require significant computational resources.
- **Complex Management**: Running and managing distributed systems can require significant expertise.

In system design, **Java Spring** (or similar frameworks) plays a critical role in building the **backend services** that interact with the frontend, data stores, and other services. Here's how frameworks like **Spring** fit into the overall architecture:

1. Backend Services (Business Logic)

Java Spring, specifically **Spring Boot**, is commonly used to create microservices that implement the business logic of an application. Here's how it fits into each layer of the system:

a. Microservices Architecture

Spring Boot simplifies building RESTful web services and microservices. It provides features like dependency injection, easy integration with databases, security, and robust configuration management.

- **Use Cases**:
  - Handling API requests from the frontend and performing business operations.
  - Interacting with databases like relational databases (MySQL, PostgreSQL) or NoSQL databases (Cassandra, DynamoDB).

    - Implementing complex business logic, validating data, and orchestrating workflows.
- **Pros**:
    - **Ease of Development**: Spring Boot automates configurations, enabling fast development.
    - **Integration**: Easily integrates with databases, messaging systems (e.g., Apache Kafka), caching mechanisms (e.g., Redis), etc.
    - **Scalability**: Supports microservices, allowing services to scale independently.
- **Cons**:
    - **Memory Usage**: Spring applications can be resource-intensive if not optimized.
    - **Learning Curve**: Understanding the Spring ecosystem and its configuration requires a learning curve, especially for new developers.

2. System to Populate Data Stores (Data Pipelines)

If the system involves **data processing or population of data stores** (like a data lake or distributed data store), Spring can work in conjunction with tools like **Kafka, Spark, or Flink** to provide a unified pipeline.

- **Spring Kafka**: Spring Kafka is used to build event-driven microservices that can produce or consume data from Apache Kafka topics.
    - **Example**: A Spring service could listen to a Kafka topic, process the data, and store it in a database.
- **Spring Batch**: Used for batch processing of large data sets, commonly in conjunction with databases, message queues, or flat files.
    - **Example**: A Spring Batch job could be used to extract data from an S3 data lake, process it, and insert it into a distributed data store like Cassandra.

3. Security Layer

Spring Security is a module that provides out-of-the-box security for Java-based applications.

- **Authentication and Authorization**: Spring Security allows you to secure APIs and microservices by handling user authentication (via OAuth2, JWT tokens) and enforcing role-based access control.
    - **Example**: Securing a RESTful API that serves frontend requests and restricts access based on user roles or permissions.
- **Cons**:
    - **Complex Configuration**: Can be complicated to configure for custom security requirements, especially in large applications.

4. Caching Layer

Spring can also handle **caching** through integrations with technologies like **Redis** or **Memcached**.

- **Spring Cache**: This abstraction allows developers to easily cache data in memory, reducing the load on databases and speeding up responses for frequently requested data.
    - **Example**: A Spring service can cache the results of expensive database queries to speed up subsequent calls to the same endpoint.
- **Pros**:
    - Reduces database load and improves performance.
    - Easy to configure with minimal changes to the codebase.
- **Cons**:
    - Cache invalidation can be tricky and needs to be managed carefully to avoid stale data.

5. Distributed Systems and Data Systems

When interacting with **distributed data systems** (e.g., **Cassandra**, **DynamoDB**), frameworks like Spring Data provide abstraction layers that simplify database interactions.

- **Spring Data**: Provides built-in abstractions for interacting with databases (both relational and NoSQL), allowing developers to work with data models more efficiently.
    - **Example**: Using Spring Data Cassandra to query, insert, or update records in a distributed database like Cassandra, abstracting away the complexity of manual query execution.
- **Pros**:
    - Simplifies database operations through a repository pattern.
    - Supports a wide range of databases (both SQL and NoSQL).
- **Cons**:
    - Performance tuning might be necessary for large-scale distributed systems.

6. APIs and Communication

Spring simplifies **API development** through its Spring MVC or Spring WebFlux modules.

- **RESTful APIs**: Spring Boot is commonly used to build REST APIs that serve as communication layers between the frontend and the backend services or microservices.
- **GraphQL**: Spring supports GraphQL for more complex API queries, especially useful when interacting with multiple data sources in one request.
- **WebSockets**: For real-time applications (such as chat apps or live dashboards), Spring can handle WebSocket connections to enable bidirectional communication between client and server.

7. Monitoring and Observability

Spring Boot integrates with monitoring tools such as **Prometheus**, **Grafana**, **Elasticsearch**, and **Kibana** for observability.

- **Spring Actuator**: Offers built-in monitoring for Spring applications, exposing metrics, health checks, and other

operational data through HTTP endpoints.
- **Distributed Tracing**: Spring supports distributed tracing with tools like Zipkin or OpenTelemetry, which is essential in microservices architectures to track requests across different services.

8. Pros and Cons of Spring in a Distributed Architecture

Pros:

- **Modular and Flexible**: Spring's modular structure allows you to choose only the components you need (e.g., Spring Boot for microservices, Spring Batch for data processing, Spring Security for security).
- **Extensive Ecosystem**: A mature ecosystem with a wide range of libraries and plugins to integrate with databases, messaging systems, caching, monitoring, and more.
- **Built for Microservices**: Spring Boot is particularly well-suited for building microservices, which are ideal for scalable and maintainable distributed systems.
- **Strong Community**: Spring has a large and active community, which makes finding resources, tutorials, and support easier.

Cons:

- **Performance Overhead**: Spring apps can become memory-heavy if not optimized, especially in microservices where there are many running instances.
- **Complexity**: The Spring ecosystem is extensive, and its configuration options can be overwhelming for new developers.
- **Tuning Required**: Requires careful tuning and configuration for high-performance distributed systems, especially for large-scale deployments.

Conclusion

In a distributed system design, **Spring** (or similar frameworks like Django, Node.js, etc.) plays a pivotal role in building and managing the **backend logic**, **APIs**, and **interaction with distributed data systems**. It handles everything from **security**, **caching**, **data processing**, and **API development** to **integration with distributed data stores** (NoSQL or SQL). By using the right mix of technologies, such as **Spring Boot** for backend services, **Cloudflare/AWS CloudFront** for CDN, and **Apache Kafka/Spring Batch** for data pipelines, you can build a highly scalable, distributed system. The choice depends on the scale, latency, and complexity of the system you're building.