Golang Interview Questions and Answers

1. Microservices & API Development

Q1: You worked on updating a microservice to support multiple tenants. How did you ensure data isolation and efficient request handling across tenants?
A: I used a **database-per-tenant** approach for strong isolation and a **shared database with row-level access control** for multi-tenancy. API requests included tenant identifiers, and middleware enforced access policies.

Q2: You optimized API responsiveness using middleware and in-memory caching. Can you explain how you structured the middleware and what caching strategies you used?
A: Middleware handled logging, authentication, and rate limiting. I used **Redis for caching**, storing frequently accessed queries and responses, reducing DB hits and API latency.

Q3: You developed APIs for data migration. What were the biggest challenges in migrating data between architectures, and how did you mitigate them?
A: Challenges included **data consistency** and **downtime minimization**. I used an **ETL pipeline**, a **blue-green deployment strategy**, and **Kafka for real-time syncs** to avoid inconsistencies.

---

2. Golang & Performance Optimization

Q4: In the **Multi-SKU Stacking** project, you built a recommendation engine. How did you optimize Goroutines and channels to handle high throughput efficiently?
A: I used a **worker pool pattern** to manage Goroutines efficiently and **buffered channels** to prevent blocking under heavy load.

Q5: You mentioned optimizing MySQL queries. Can you provide an example of a query optimization technique you applied and how it impacted performance?
A: I used **indexing**, optimized queries with **EXPLAIN ANALYZE**, and **batch inserts** to reduce query execution time by 40%.

Q6: How do you handle memory management in Golang when dealing with large datasets? Have you ever encountered performance bottlenecks related to garbage collection?
A: I use **sync.Pool for object reuse**, minimize **pointer allocations**, and monitor memory using pprof to optimize GC behavior.

---

3. Kafka & Event-Driven Architecture

Q7: In the **Multi-SKU Stacking** project, you worked with Kafka for message processing. How did you ensure event ordering and avoid message duplication?
A: I ensured **partitioning by key (SKU_ID)** for ordering and used **Kafka transactions and idempotent consumers** to avoid duplication.

Q8: How do you handle backpressure when consuming messages from Kafka in a Golang microservice?
A: I used **rate-limited consumers**, **batch processing**, and **dead-letter queues (DLQ)** to prevent message loss.

Q9: Have you implemented a **dead-letter queue (DLQ)** in Kafka? If so, how does it improve fault tolerance in distributed systems?
A: Yes, DLQ captures **failed messages** for later analysis, preventing **system crashes and message loss**.

---

4. Kubernetes, Docker & Deployment

Q10: You reduced microservice costs by optimizing pod downsizing. What strategies did you use for right-sizing pods, and how did you monitor their performance?
A: Used **HPA (Horizontal Pod Autoscaler)**, configured resource limits, and monitored usage with **Prometheus and Grafana**.

Q11: In Kubernetes, how do you handle **rolling updates** and ensure zero downtime for a critical microservice?
A: Used **readiness probes**, **canary deployments**, and **blue-green strategies** to roll out updates gradually.

Q12: How would you debug a Golang microservice running in Kubernetes that is intermittently failing under load?
A: Used **kubectl logs, Prometheus metrics, and distributed tracing** with OpenTelemetry to pinpoint bottlenecks.

---

5. System Design & Best Practices

Q13: If you were designing a high-availability Golang-based microservice from scratch, how would you structure it?
A: I'd use **event-driven architecture**, **CQRS pattern**, **distributed caching**, and **auto-scaling with Kubernetes**.

Q14: How do you handle **distributed transactions** in a microservices architecture when multiple services update different databases?
A: Used **SAGA pattern** with compensating transactions to maintain data consistency across services.

Q15: How would you implement **rate-limiting** in a Golang API to prevent abuse while ensuring a good user experience?
A: Used **token bucket or leaky bucket algorithms** with Redis-based **rate-limiting middleware** in Gin.

This document compiles **tailored Golang interview questions and answers** based on real-world projects and experience. 🚀