Golang Microservices & Performance Optimization Q&A

1. Microservices in Golang

Q: Can you walk me through the architecture of a Golang-based microservice you've built?
A:
I developed a Golang-based microservice that handled multi-tenant support for a large-scale application. The architecture followed a clean separation of concerns with the following layers:

- **API Layer:** REST-based endpoints using net/http or gin-gonic.
- **Service Layer:** Business logic encapsulated in service structs.
- **Persistence Layer:** Used gorm as the ORM for MySQL with optimized queries.
- **Message Queue:** Used Kafka for event-driven communication between microservices.
- **Caching:** Implemented Redis for reducing database hits and improving response times.
- **Observability:** Integrated Prometheus for monitoring and OpenTelemetry for tracing.

This architecture ensured scalability and maintainability, allowing seamless tenant onboarding.

Q: How do you manage inter-service communication? Do you use gRPC, REST, or something else?
A:
It depends on the use case:

- For **synchronous communication**, I use **gRPC** because of its low latency and efficient protobuf serialization.
- For **public-facing APIs**, I use **REST** with OpenAPI specifications.
- For **asynchronous communication**, **Kafka** helps decouple services and ensures reliability.

Example: A **recommendation engine** microservice used Kafka to publish SKU stacking events, which were consumed by a planogram service.

---

2. Concurrency and Performance

Q: How do you optimize Goroutine usage in a high-throughput application?
A:

- **Worker Pools:** Instead of launching a goroutine per request, I use a worker pool pattern to limit the number of concurrent goroutines.
- **Buffered Channels:** To avoid blocking, I use buffered channels for controlling load.
- **sync.Pool:** Reuses objects instead of creating new ones, reducing GC overhead.
- **Profiling with pprof:** Identifies bottlenecks to optimize Goroutine usage.

Example: In a **real-time SKU stacking service**, I used a worker pool to process stacking recommendations efficiently.

Q: Can you explain how Golang handles memory management and garbage collection?
A:

- Golang uses a **tricolor mark-and-sweep garbage collector**, which runs concurrently to avoid stop-the-world pauses.
- The GC frequency depends on heap allocations, and memory management is done via **stack vs. heap allocation**.
- Techniques to reduce GC overhead include:
    - Using sync.Pool for object reuse.
    - Avoiding excessive pointer dereferencing.
    - Profiling allocations using pprof to detect memory leaks.

---

3. Kafka Integration

Q: How have you used Kafka in your projects?
A:
In a **multi-SKU stacking system**, Kafka helped with:

- **Event-driven processing**: SKU updates were published as events instead of API calls.
- **Scalability**: Producers and consumers scaled independently.
- **Data consistency**: Used Kafka transactions for atomic event processing.

Q: How do you ensure message ordering and exactly-once processing in a distributed system?
A:

1. **Ordering:**
    - Kafka ensures ordering within a partition. I use **partition keys** (e.g., SKU_ID) to maintain sequence.
2. **Exactly-once Processing:**
    - Enable **idempotency** in Kafka producers.
    - Use a **transactional producer** and **consumer group offsets** to avoid duplicate processing.
    - Implement **deduplication mechanisms** at the application level.

---

4. Database Optimization

Q: What techniques have you used to optimize MySQL queries in a Golang application?
A:

- **Indexing:** Added composite indexes to speed up WHERE clauses.
- **Query Optimization:** Used EXPLAIN ANALYZE to optimize SQL queries.
- **Connection Pooling:** Used sql.DB with SetMaxOpenConns and SetConnMaxLifetime for optimal performance.

- **Caching:** Used Redis to store frequently accessed queries, reducing DB hits.
- **Batch Processing:** Reduced DB load by processing inserts/updates in bulk instead of row-by-row.

Q: Have you worked with any ORM libraries in Golang? If so, which one and why?
A:
Yes, I've used gorm because:

- It supports **query building** and **transactions**.
- It provides **hooks and callbacks** for data validation.
- Auto-migrations simplify schema evolution.

However, for performance-sensitive use cases, I prefer using raw SQL with database/sql.

---

5. Containerization and Deployment

Q: Can you describe your experience with deploying Golang applications using Kubernetes and Docker?
A:

1. **Dockerization:**
    - Used scratch or alpine images to reduce image size.
    - Multi-stage builds to keep binaries lightweight.
2. **Kubernetes Deployment:**
    - Used **Helm charts** to manage deployments.
    - Configured **HPA (Horizontal Pod Autoscaler)** for scaling based on CPU/memory.
    - Used **liveness and readiness probes** for health checks.

Q: How do you handle service discovery and scaling in Kubernetes?
A:

1. **Service Discovery:**
    - Used **Kubernetes Services (ClusterIP, NodePort, LoadBalancer)**.
    - For inter-service communication, used **DNS-based service discovery** (service-name.namespace.svc.cluster.local).
2. **Scaling:**
    - **Horizontal Scaling**: HPA automatically increases replicas based on load.
    - **Vertical Scaling**: Adjusted resource requests/limits in YAML.
    - Used **metrics-server and Prometheus** to monitor scaling efficiency.

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