Name : Assad Wazeer

Reg No:FA22-BSE-085 (BSE-5A)

---

## 1. Scalability Issues in Monolithic Architecture

**Problem:**

- In a monolithic architecture, the entire application is built as a single unit with a single codebase.
- As the application grows, the codebase becomes large and complex, making deployments challenging.
- Scaling is inefficient because you must scale the entire application even if only one component needs additional resources.

**Solution:**

- **Migrate to Microservices Architecture**:
  - Break down the monolith into smaller, independently deployable services (microservices).
  - Each microservice is responsible for a specific functionality (e.g., user management, order processing).
  - Microservices communicate via lightweight protocols (e.g., REST, gRPC).
- **Advantages**:
  - Independent scaling of services based on their specific requirements.
  - Easier deployments and faster development cycles.
  - Improved fault isolation since the failure of one microservice doesn't bring down the entire system.

**Tools and Techniques:**

- **Containerization**: Use Docker and Kubernetes to deploy microservices.
- **Service Mesh**: Tools like Istio or Linkerd can manage communication between microservices.

---

## 2. Performance Bottlenecks in Database Layer

**Problem:**

- As data grows, queries become slower, and the database struggles to handle the load.

- High latency affects the application's performance and user experience.
- A single database may become a bottleneck for read and write operations.

**Solution:**

- **Database Sharding**:
  - Split the database into smaller, independent parts (shards) based on specific criteria (e.g., user ID).
  - Each shard is stored on a different server, distributing the load.
- **Caching Layers**:
  - Use caching systems like Redis or Memcached to store frequently accessed data in memory.
  - Reduces the need to query the database repeatedly.
- **Read Replicas**:
  - Create replicas of the database for read operations, distributing the read workload across multiple servers.
  - Use the primary database for writes and synchronize changes with replicas.

**Tools and Techniques:**

- **Partitioning**: Divide large tables into smaller, more manageable partitions.
- **Query Optimization**: Use proper indexing and analyze query performance with tools like `EXPLAIN`.

---

## 3. High Coupling Between Components

**Problem:**

- When components in a system are tightly coupled, changes in one module cascade into others.
- This increases the maintenance burden and makes the system brittle, as a small change can lead to system-wide issues.

**Solution:**

- **Apply SOLID Principles**:
  - **Single Responsibility Principle**: Each module or class should have one reason to change.
  - **Open/Closed Principle**: Classes should be open for extension but closed for modification.
  - **Dependency Inversion Principle**: Depend on abstractions rather than concrete implementations.
- **Use Dependency Injection**:
  - Pass dependencies into components instead of hardcoding them.

    - This allows for easier testing and replacement of components without modifying the code.

**Tools and Techniques:**

- Use frameworks like Spring (Java) or NestJS (Node.js) to manage dependency injection.
- Design systems using clean architecture patterns (e.g., hexagonal or onion architecture).

---

## 4. Poor Fault Tolerance

**Problem:**

- A single point of failure (SPOF) can cause system-wide outages if a critical component fails.
- Lack of redundancy or fallback mechanisms can result in unavailability or data loss.

**Solution:**

- **Implement Circuit Breakers**:
    - Use circuit breaker patterns to detect failures and prevent cascading issues.
    - For example, if a downstream service is unavailable, the circuit breaker opens and prevents further requests to that service.
- **Fallback Mechanisms**:
    - Provide alternative solutions in case of failure (e.g., default responses, cached data).
- **Redundancy Across Multiple Availability Zones**:
    - Deploy services and databases across multiple availability zones or regions to ensure high availability.
    - Use load balancers to distribute traffic across redundant resources.

**Tools and Techniques:**

- **Resilience Libraries**: Use libraries like Netflix Hystrix or Spring Cloud Resilience4j.
- **Cloud Infrastructure**: AWS, Azure, and Google Cloud provide native redundancy options.

---

## 5. Security Vulnerabilities in API Architecture

**Problem:**

- APIs without proper authentication and authorization controls are vulnerable to attacks like unauthorized access, data breaches, and rate-based abuse.

- Exposure to denial-of-service (DoS) attacks due to lack of rate limiting.

**Solution:**

- **Implement OAuth 2.0**:
  - Use OAuth 2.0 for secure authentication and authorization.
  - For example, users can log in using third-party providers like Google or Facebook.
- **API Gateways**:
  - Use API gateways (e.g., Kong, Apigee, AWS API Gateway) to centralize security and rate-limiting policies.
  - Gateways can filter requests, apply security measures, and monitor traffic.
- **Rate Limiting**:
  - Limit the number of requests per user or client in a given timeframe to prevent abuse.
  - Use tools like NGINX or Envoy for rate limiting.

**Tools and Techniques:**

- **JWT (JSON Web Tokens)**: Securely transmit authentication data.
- **API Monitoring**: Use tools like Datadog or Splunk to detect suspicious activities.

---

## Conclusion

Addressing these architectural problems ensures a scalable, maintainable, and secure software system. Here's a quick summary:

| Problem | Solution | Benefits |
|---|---|---|
| **Scalability Issues** | Migrate to microservices | Independent scaling, faster deployments, fault isolation |
| **Performance Bottlenecks** | Database sharding, caching, and read replicas | Faster query response, distributed load |
| **High Coupling** | Apply SOLID principles and dependency injection | Easier maintenance, modularity |
| **Poor Fault Tolerance** | Circuit breakers, fallbacks, and redundancy | High availability, resilience to failures |
| **Security Vulnerabilities** | OAuth 2.0, API gateways, and rate limiting | Improved API security, prevention of abuse, and unauthorized access |