

### **176. What is the SAGA pattern?**

A distributed transaction pattern where each step is a local transaction with a compensating step for rollback.

### **177. Types of SAGA implementations?**

- Choreography: Events trigger service actions
- Orchestration: Central coordinator controls flow

### **178. When should you use the SAGA pattern?**

Use it when you need consistency across services without 2PC, and when scalability is a concern.

### **179. What is a compensating transaction?**

An undo operation for a previously successful step in a distributed transaction flow.

### **180. SAGA vs 2PC (Two-Phase Commit)**

SAGA: Event-driven, scalable, loose coupling

2PC: Blocking, tightly coupled, less scalable

### **181. What is Eventual Consistency?**

A system design approach where consistency is achieved over time, not instantly.

### **182. Techniques for implementing eventual consistency?**

- Event-driven architecture
- SAGA pattern
- Retry and reconciliation services

### **183. What is a message broker and why use it?**

Middleware like Kafka/RabbitMQ that enables decoupled, asynchronous communication between services.

### **184. What is Domain-Driven Design (DDD)?**

An approach to model software around the business domain with concepts like bounded contexts and aggregates.

### **185. What is a bounded context in DDD?**

A logical boundary where a particular model is valid. Helps in isolating terms and models in complex domains.

### **186. How does DDD support microservices?**

Each bounded context maps to a microservice. Enables decoupling and clear domain modeling.

### **187. What is an aggregate in DDD?**

A group of domain objects with a single entry point (aggregate root) ensuring consistency within the group.

### **188. What is eventual consistency within an aggregate?**

Consistency is maintained within an aggregate; across aggregates it's eventual, typically via events.

### **189. What is the Strangler Fig Pattern?**

Gradually replaces a monolith by routing new features to microservices while deprecating old parts.

### **190. What is the Backends for Frontends (BFF) pattern?**

Custom backends built for each UI (web, mobile) to serve optimized data for that frontend.

### **191. What is an anti-corruption layer (ACL)?**

A translation layer to keep legacy systems from influencing the new domain model.

### **192. What is Polyglot Persistence?**

Using different databases (SQL, NoSQL, etc.) for different microservices depending on their needs.

### **193. What is CAP Theorem?**

You can pick two out of: Consistency, Availability, Partition Tolerance. Partition Tolerance is usually mandatory.

### **194. CAP theorem in context of microservices**

Microservices often favor Availability and Partition Tolerance, trading off strict consistency.

### **195. What is eventual consistency vs strong consistency?**

Eventual: Updates propagate over time

Strong: All nodes always see the same data instantly

### **196. What is API composition?**

A pattern where an aggregator fetches and combines data from multiple services for a response.

### **197. Drawbacks of API composition?**

- High latency
- Tight coupling
- Difficult to scale with complex workflows

### **198. What is CQRS + Event Sourcing combo?**

Commands produce events that are stored and replayed to build read models. Enables full audit and separation of models.

### **199. What is eventual consistency in messaging systems like Kafka?**

Messages are delivered with delay; idempotent consumers help retry safely without duplication.

### **200. Final best practices for advanced microservices?**

- Use SAGA for distributed consistency
- DDD for design
- Event-driven architecture
- Secure and observe everything