Saga Pattern

*The Saga architecture pattern provides transaction management using a sequence of local transactions.*

*A local transaction is the unit of work performed by a Saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction. Further, the Saga pattern guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed.*

*In the Saga pattern, a compensating transaction must be idempotent and retryable. These two principles ensure that we can manage transactions without any manual intervention.*

***Choreography –***

*In choreography pattern, all microservices publish an event that is processed by next microservices.*

*In the Saga, choreography flow is successful if all the microservices complete their local transaction, and none of the microservices reported any failure.*

*In the event of a failure, the microservice reports the failure to SEC, and it is the SEC’s responsibility to invoke the relevant compensation transactions.*

*If the call to the compensating transaction fails, it is the SEC’s responsibility to retry it until it is successfully completed.*

*The Choreography pattern works for greenfield microservice application development. Also, this pattern is suitable when there are fewer participants in the transaction.*

***Saga Orchestration –***

*In the Orchestration pattern, a single orchestrator is responsible for managing the overall transaction status.*

*If any of the microservices encounter a failure, the orchestrator is responsible for invoking the necessary compensating transaction.*

*The Saga orchestration pattern is useful for brownfield microservice application development architecture. In other words, this pattern works when we already have a set*

*of microservices and would like to implement the Saga pattern in the application. We need to define the appropriate compensating transactions to proceed with this pattern.*

*Alerts-*

*you should configure monitoring and alerting for your services to be sure that system is alive and able to process messages.*

*Also you should consider whether you need to implement some compensation logic or checks. Imagine: you make two HTTP POST requests to different services while processing a message, 1st service call is completed successfully but 2nd one fails. If you retry entire CreateOrder message - you should not call the 1st service again.*

*In order for the communication to be reliable, it’s essential that the saga participants use a message broker that guarantees at-least-once delivery and has durable subscriptions. completes even if a participant is temporarily unavailable.*

1. Services vs Serverless  
   Mostly following this pattern means having a business logic that spreads across many services. However, there are specific situations when not all the business steps require execution or only a few steps are necessary. Should these steps be deployed as functions instead of services in these scenarios? Events usually trigger functions, which shut down once they complete their job. Such an infrastructure can save us money compared to a service that remains active continuously and performs minimal tasks.
2. Recovery from Transient Failures  
   The orchestration pattern implementation can be challenging because it involves coordinating multiple services and workflows, which requires a different approach to designing and managing software systems than traditional monolithic architectures. The implementation must be able to handle potential transient failures, such as network failure, service failure, or database failure. Below are a few ways to cater to such issues:
   * Retry Mechanism  
     Implementing a retry mechanism can improve resiliency when a service operation fails. The retry mechanism should configure the number of retries allowed, the delay between retries, and the conditions to attempt retries.
   * Circuit Breaker Pattern  
     In case a service fails, the orchestrator must detect the failure, isolate the failed service, and give it a chance to recover. It can help the service heal without disruption and avoid complete system failure.
   * Graceful Degradation  
     If a service fails and becomes unavailable, the rest of the services should continue to operate. The orchestrator should look for fallback options to minimize the impact on end-users, such as previously cached results or an alternate service.
3. Monitoring and Alerting  
   The entire business flow is distributed among various services when we operate with the Orchestration Pattern. Therefore, an effective monitoring and alerting solution is mandatory to trace and debug any failures. The solution must be capable of detecting any issues in real time and taking appropriate actions to mitigate the impact. It includes implementing auto-recovery strategies, such as restarting failed services or switching to a backup service, and setting up alerts to notify the operations team when exceptions occur. The logs generated by the orchestrator are also valuable for the operations team to troubleshoot errors. We can operate smoothly and meet user needs by proactively identifying and resolving issues.
4. Orchestration Service Failure  
   Finally, we must prepare for scenarios where the orchestrator fails itself while processing requests. For instance, in our payment gateway example, imagine a scenario where the orchestrator calls the Transaction service to transfer the funds but crashes or loses connection before getting a successful response for the occurred transaction. It could lead to a frustrating user experience, with the risk of the customer being charged twice for the same product. To prevent such failure scenarios, we can adopt one of the following solutions:
   * Service Replication  
     Replicate the orchestration service across multiple nodes. The service can automatically failover to the backup node when needed. With a load balancer that can detect and switch to the available node, the replication guarantees seamless service and prevents disruptions to the user.
   * Data Replication  
     Not only should we replicate the service, but we should also replicate the data to ensure data consistency. It enables the backup node to take over seamlessly without any data loss.
   * Request Queues  
     Implementing queues functions like a buffer for requests when the orchestration service is down. The queue can hold incoming requests until the service is available again. Once the backup node is up and running, it can retrieve the data from the queue buffer and process them in the correct order.

*2PC: Save Customer for every received Invoice request, while both are managed by 2 different systems.*

*Sagas: Book a flight itinerary consisting of several connecting flights, while each individual flight is operated by different airlines.*