Database Patterns

MOSTAFA RASTGAR

AUGUST 2019

Agenda

- Database Patterns
  - Database per Service
  - Shared Database per Service
  - CQRS
  - Saga Pattern
- Review

Database per Service

- Problem
  - There is a problem of how to define database architecture for microservices. Following are the concerns to be addressed:
    1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.
    2. Business transactions may enforce invariants that span multiple services.
    3. Some business transactions need to query data that is owned by multiple services.
    4. Databases must sometimes be replicated and sharded in order to scale.
    5. Different services have different data storage requirements.Solution
- Solution
  - To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly.
  - For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service. Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

Shared Database per Service

- Problem
  - We have talked about one database per service being ideal for microservices, but that is possible when the application is greenfield and to be developed with DDD. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. What is the suitable architecture in that case?
- Solution
  - A shared database per service is not ideal, but that is the working solution for the above scenario.
  - Most people consider this an anti-pattern for microservices, but for brownfield applications, this is a good start to break the application into smaller logical pieces.
  - This should not be applied for greenfield applications.
  - In this pattern, one database can be aligned with more than one microservice, but it has to be restricted to 2-3 maximum, otherwise scaling, autonomy, and independence will be challenging to execute.

Command Query Responsibility Segregation (CQRS)

- Problem
  - Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services
  - it's not possible. Then, how do we implement queries in microservice architecture?
- Solution
  - CQRS suggests splitting the application into two parts
    - The command side handles the Create, Update, and Delete requests.
    - The query side handles the query part by using the materialized views.
  - The event sourcing pattern is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

CQRS Example

- In our payment project we should do the following issues:
  - After successfully checking available balance in balace-check microservice, we should persist the payment order with not_sent status
  - After sending it in response-sender microservice we should update its status to the sent
  - Another microservice, should periodically retrieve the not_sent payment orders to try resending
  - Afterwards, the successfully sent payment orders' status should be updated to sent
- CQRS Solution:
  - paymentorder-commands microservice for payment orders creation and manipulation
  - resending-service micoroservice for retrying

Saga Pattern

- Problem
  - When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services?
  - For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer's credit limit.
  - Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.
- Solution
  - A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:
    - Choreography — When there is no central coordination, each service produces and listens to another service's events and decides if an action should be taken or not.
    - Orchestration — An orchestrator (object) takes responsibility for a saga's decision making and sequencing business logic.
  - Axon framework has a good integration with spring boot. For more detail, refer: http://progressivecoder.com/saga-pattern-implementation-axon-spring-boot-part-3/

Saga Example

- In balance-check microservice:
  - first, we can save the payment order using paymentorder-commands microservice
  - Then try to send it to the banks topics using response-sender microservice
  - If something went wrong here, a compensating request could be deleting the payment order using paymentorder-commands microservice as well as depositing the available balance using balance-check microservice
  - As a result of this, the status field in payment order as well as the entire resending-service micoroservice could be eliminated

Review

- Question
  - How to break the application into smaller pieces
- Answer
  - DDD - Domain Driven Design
  - Bounded Context, Domain, Sub-Domain

Review

- Question
  - Configuration of several spring beans would be a struggling activity. How to ease the problem?
- Answer
  - Spring boot provides basic configuration needed to configure the application with the applied frameworks.
  - It is called Auto Configuration

Review

- Question
  - How should we know whether we added proper dependencies for a specific framework such as Spring Data?
- Answer
  - Each of which has an appropriate Starter including a set of convenient dependency descriptors that you can include in your application.

Review

- Question
  - How can we find inner microservices address in development and production environment?
- Answer
  - Eureka Service Discovery

Review

- Question
  - How can we have an optimal call among several instances of a single microservice
- Answer
  - Netflix ribbon is a convenient client-side load-balancer
  - Netflix Zuul is simultaneously an API Gateway and centralized load-balancer
    - Internally, Zuul uses Netflix Ribbon to look up for all instances of the service from the service discovery (Eureka Server)

Review

- Question
  - How can we queue several concurrent requests coming from our rest endpoints, load-balancing them and fully support failover?
- Answer
  - Apache Kafa using Apache Zookeeper provide such capabilities with some amazing mechanisms such as topics, partitioning, leader election, consumer group, etc.

Review

- Question
  - while deploying new release in production, if we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business. Besides, the rollback will be a nightmare
- Answer
  - Apply Blue-Green or Canary deployment pattern

Review

- Question
  - For each environment like dev, QA, prod, the endpoint URL or some configuration properties might be different. How can we automatically change the configurations based on our environment?
- Answer
  - Externalize all the configuration, including endpoint URLs and credentials using configuration server
  - Moreover you can store your secret data in a Vault server
  - or encrypt it using either a secret key or pair key and in configuration server, they will be automatically decrypted in runtime

Review

- Question
  - How can we harmonize the logs generated in several microservice?
- Answer
  - Log Aggregation design pattern
  - Kafa + ELK could be a general best-practice solution

Review

- Question
  - When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions. How should we collect metrics to monitor application perfomance?
- Answer
  - Performance Metrics design pattern
  - Prometheus + Spring Boot Actuator

Review

- Question
  - When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. In that case, how do you ensure a request doesn't go to those failed instances?
- Answer
  - Health Check design pattern
  - Spring Boot Actuator does implement a /health endpoint and the implementation can be customized, as well

Review

- Question
  - What will happen if downstream service is down? Can we skip it and whenever it comes back to normal state, we engage it again?
- Answer
  - Circuit breaker design pattern
  - Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips
  - Hystrix Dashboard is a continent tool to monitor the circuit

Review

- Question
  - How can we monitor the time spans of several process in our microservices?
- Answer
  - Twitter has a convenient tool called zipkin for tracing
  - Spring Sleuth can send trance info to zipkin

Review

- Question
  - Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services. What should we do in microservices?
- Answer
  - CQRS design pattern

Review

- Question
  - When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services?
- Answer
  - Saga design pattern

Congrats,