Week 2

DB Roles in Microservice Architecture

IS457 - Advanced Database Management

THAMMASAT BUSINESS SCHOOL

AACSB ACCREDITED

EFMD EQUIS ACCREDITED

ASSOCIATION OF MBAS ACCREDITED

Outline

- Online Materials: https://mentor2code.com/courses/is457/
- Exercise - SQL vs NoSQL
    - https://mentor2code.com/courses/is457/week-01.html
- Microservice Architecture

Monolithic Architecture

- The large monolithic code base intimidates developers.
- Overloaded IDE and web container.
- Difficult for development, deployment, and scaling.
- Requires a long-term commitment to a technology stack.

Monolithic Architecture available at https://microservices.io/patterns/monolithic.html

Microservice Architecture

- Enables the continuous delivery and deployment of large, complex applications.
- Each microservice is relatively small.

Microservice Architecture available at https://microservices.io/patterns/microservices.html

Microservice Architecture - Drawback

- Developers must deal with the additional complexity of creating a distributed system:
  - Implementing the **inter-service communication** mechanism and dealing with partial failure.
  - Implementing **requests that span multiple services** is more difficult.
  - **Testing** the interactions between services is more difficult.
  - Implementing requests that span multiple services requires careful **coordination between the teams**.
  - Developer **tools/IDEs** are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
- Deployment complexity.

Microservice Architecture available at https://microservices.io/patterns/microservices.html

Shared Database

- Use a (single) database that is shared by multiple services.
- Each service freely accesses data owned by other services using local ACID transactions.

```
BEGIN TRANSACTION
…
SELECT ORDER_TOTAL
 FROM ORDERS WHERE CUSTOMER_ID = ? (Locked)
…
SELECT CREDIT_LIMIT
FROM CUSTOMERS WHERE CUSTOMER_ID = ? (Locked)
…
INSERT INTO ORDERS …
…
COMMIT TRANSACTION (Released)
```

Shared database available at https://microservices.io/patterns/data/shared-database.html

Shared Database

- **Benefits:**
  - A developer uses **familiar and straightforward ACID transactions** to enforce data consistency.
  - A single database is **simpler to operate.**
- **Drawbacks:**
  - **Development time coupling** - a developer working on, for example, the OrderService will need to coordinate schema changes with the developers of other services that access the same tables. This coupling and additional coordination will slow down development.
  - Runtime coupling - because all services access the same database they can **potentially interfere** with one another. For example, if long running CustomerService transaction holds a lock on the ORDER table then the OrderService will be blocked.
  - **Single database might not satisfy** the data storage and access requirements of all services.

Shared database available at https://microservices.io/patterns/data/shared-database.html

Database per Service

- Keep each microservice's persistent data private to that service and **accessible only via its API**.
- A service's transactions only involve its database.
- The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

Database per service available at https://microservices.io/patterns/data/database-per-service.html

Database per Service

- There are a few different ways to keep a service's persistent data private.
  - **Private-tables-per-service –** each service owns a set of tables that must only be accessed by that service.
  - **Schema-per-service –** each service has a database schema that's private to that service.
  - **Database-server-per-service –** each service has it's own database server.
- Private-tables-per-service and schema-per-service have the **lowest overhead.**
- Using a schema per service is appealing since it **makes ownership clearer**.
- Some **high throughput services** might need **their own database server.**

Database per service available at https://microservices.io/patterns/data/database-per-service.html

Database per Service

- **Benefits:**
  - Helps ensure that the services are loosely coupled. **Changes** to one service's database **does not impact** any other services.
  - Each service can use the **type of database that is best suited to its needs.** For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.
- **Drawbacks:**
  - Implementing business transactions that span multiple services is not straightforward. **Distributed transactions are best avoided** because many modern (NoSQL) databases don't support them.
  - Implementing queries that join data that is now in multiple databases is challenging.
  - Complexity of **managing multiple SQL and NoSQL databases**.

Database per service available at https://microservices.io/patterns/data/database-per-service.html

Database per Service

- Patterns/solutions for implementing transactions and queries that span services:
- Implementing **transactions** that span services - use the **Saga pattern.**
- Implementing **queries** that span services:
  - **API Composition -** the application performs the join rather than the database.
  - **Command Query Responsibility Segregation (CQRS) -** maintain one or more materialised views that contain data from multiple services.

Database per service available at https://microservices.io/patterns/data/database-per-service.html

Saga

- Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services.
- For example, lets imagine that you are building an e-commerce store 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.

Saga at https://microservices.io/patterns/data/saga.html

Saga

Saga at https://microservices.io/patterns/data/saga.html

Saga

- Two ways of coordination sagas:
  - **Choreography** - each local transaction publishes domain events that trigger local transactions in other services.
  - **Orchestration** - an orchestrator (object) tells the participants what local transactions to execute.
- **Benefits:** It enables an application to maintain data consistency across multiple services without using distributed transactions.
- **Drawbacks:** The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga
- **Issues to address:** In order to be reliable, a service must atomically update its database and publish a message/event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker.

Saga at https://microservices.io/patterns/data/saga.html

API Composition

- Implement a query by defining an API Composer, which invoking the services that own the data and performs an in-memory join of the results.
- It a simple way to query data in a microservice architecture.
- Some queries would result in inefficient, in-memory joins of large datasets.

API Composition available at https://microservices.io/patterns/data/api-composition.html

Command Query Responsibility Segregation (CQRS)

- How to implement a query that retrieves data from multiple services in a microservice architecture?
- Define a view database, which is a read-only replica that is designed to support that query.
- The application keeps the replica up to data by subscribing to Domain events published by the service that own the data.

CQRS available at https://microservices.io/patterns/data/cqrs.html

CQRS

- **Benefits:**
  - Supports multiple denormalised views that are scalable and performant.
  - Improved separation of concerns = simpler command and query models.
  - Necessary in an event sourced architecture.
- **Drawbacks:**
  - Increased complexity.
  - Potential code duplication.
  - Replication lag/eventually consistent views.

CQRS available at https://microservices.io/patterns/data/cqrs.html