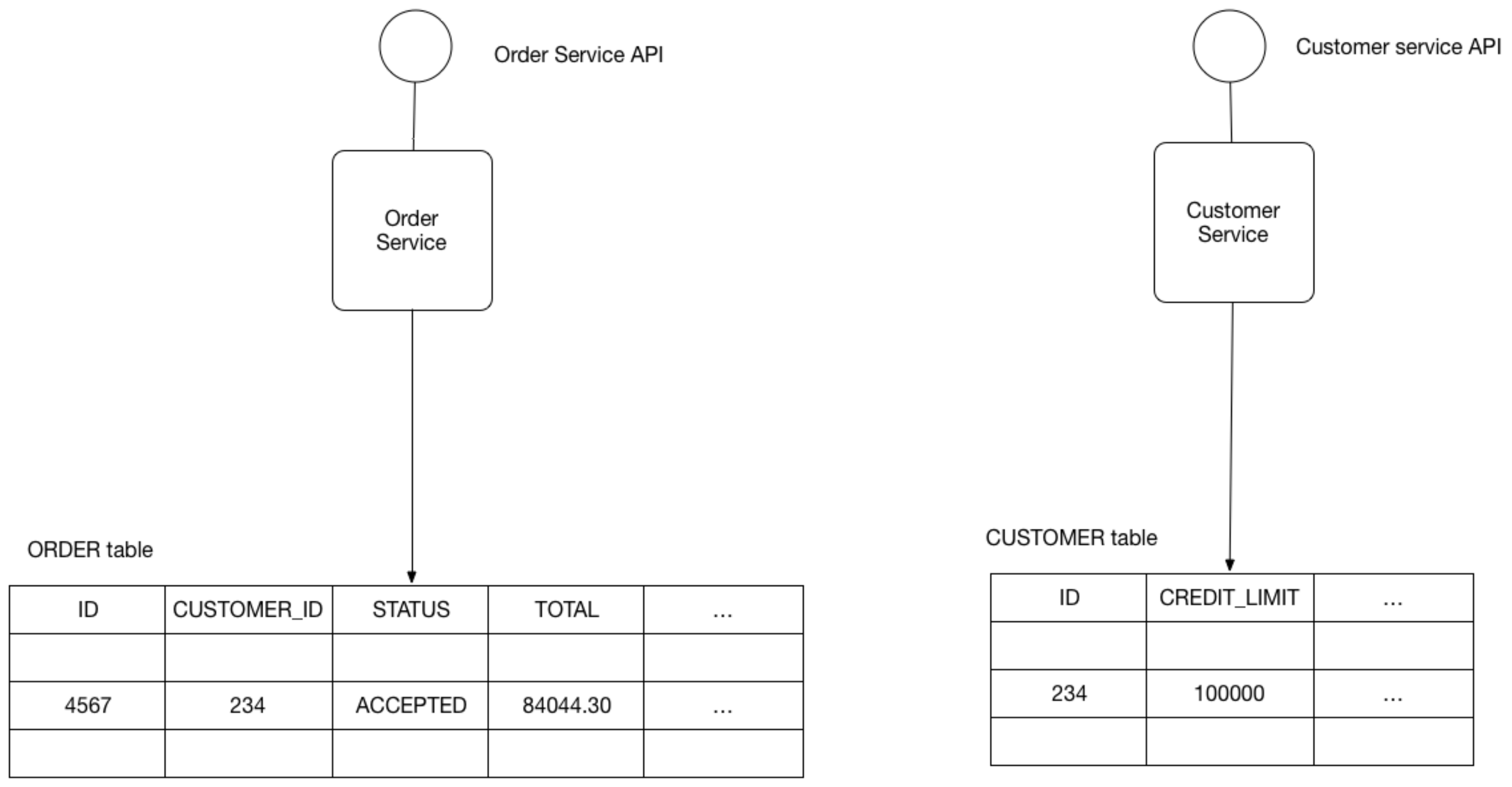
Pattern: Shared database

Context

Let's imagine you are developing an online store application using the Microservice architecture pattern (/patterns/microservices.html). Most services need to persist data in some kind of database. For example, the `Order Service` stores information about orders and the `Customer Service` stores information about customers.



ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

ID	CREDIT_LIMIT	...
234	100000	...

Problem

What's the database architecture in a microservices application?

Forces

- Services must be loosely coupled so that they can be developed, deployed and scaled independently
- Some business transactions must enforce invariants that span multiple services. For example, the `Place Order` use case must verify that a new Order will not exceed the customer's credit limit. Other business transactions, must update data owned by multiple services.
- Some business transactions need to query data that is owned by multiple services. For example, the `View Available Credit` use must query the Customer to find the `creditLimit` and Orders to calculate the total amount of the open orders.
- Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.
- Databases must sometimes be replicated and sharded in order to scale. See the Scale Cube (/articles/scalecube.html).

- Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

Solution

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

Example

The `OrderService` and `CustomerService` freely access each other's tables. For example, the `OrderService` can use the following ACID transaction ensure that a new order will not violate the customer's credit limit.

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

The database will guarantee that the credit limit will not be exceeded even when simultaneous transactions attempt to create orders for the same customer.

Resulting context

The benefits of this pattern are:

- A developer uses familiar and straightforward ACID transactions to enforce data consistency
- A single database is simpler to operate

The drawbacks of this pattern are:

- 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.

Related patterns

- Database per Service (database-per-service.html) is an alternative approach

Tweet