Microservices With CQRS and Event Sourcing

by Kiran Kumar · Feb. 18, 19 · Microservices Zone · Tutorial

The main topic of this article is to describe how we can integrate an event-driven architecture with microservices using event sourcing and CQRS.

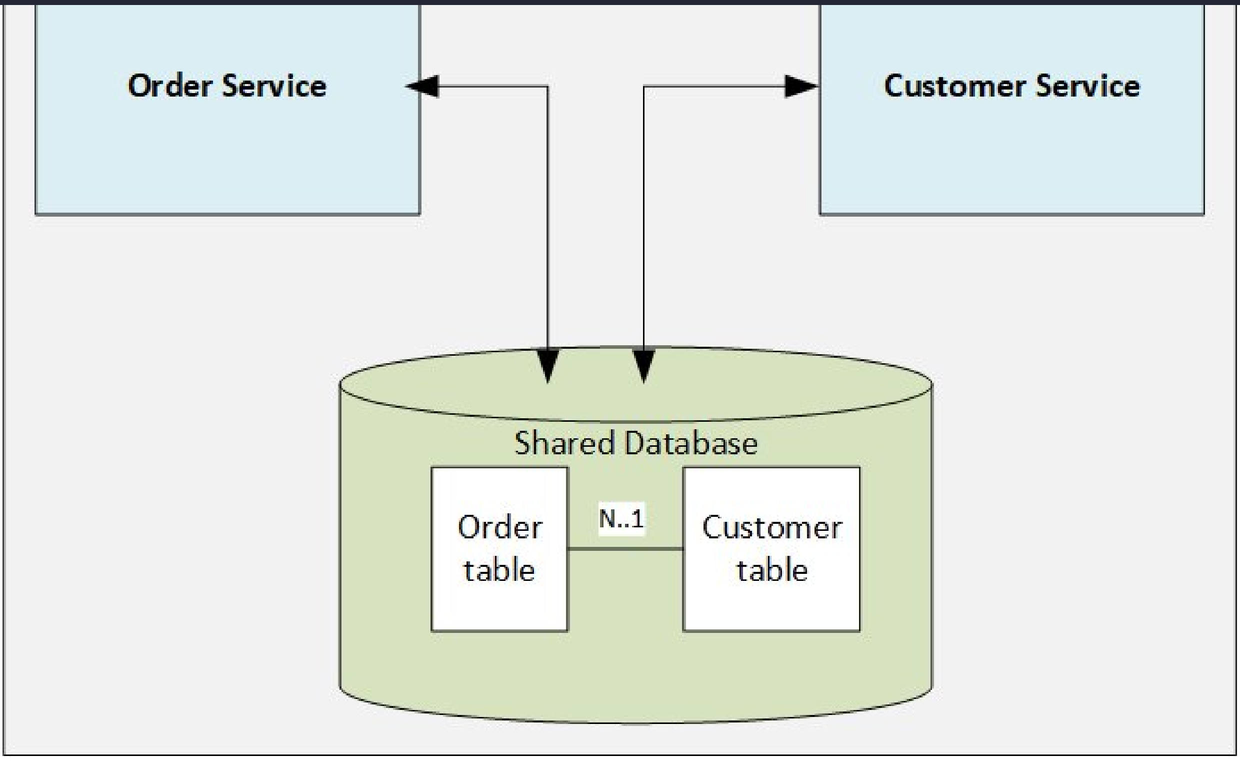
Microservices are independent, modular services that have their own layered architecture.

When microservices share the same database, the data model among the services can follow relationships among the tables associated with the microservices.

For example, there are two microservices running in their own containers: 'Order' and 'Customer.'

The Order service will take care of creating, deleting, updating, and retrieving order data. The Customer service will work with customer data.

One customer can have multiple orders, which has a one-to-many relationship. As both tables are in a single database, a one-to-many relationship can be established.



The are some limitations to this approach, however. A shared database is not recommended in a microservices-based approach, because, if there is a change in one data model, then other services are also impacted.

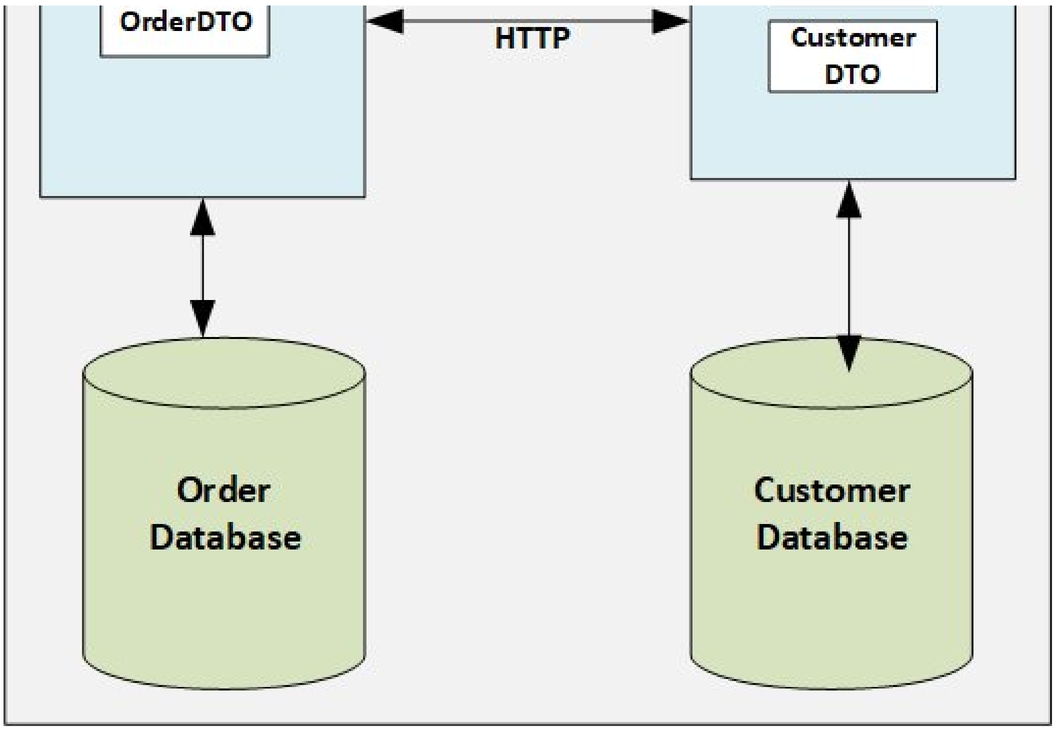
As part of microservices best practices, each microservice should have its own database.

The Order microservice access the Order database and the Customer microservice access the Customer database.

In this scenario, the relationships among the tables cannot be established, as both tables are in separate databases.

If the Customer microservice wants to update the Order data, the Customer microservice can pass teh customer id as a request parameter to the HTTP service of the Order microservice to update the Order data for the corresponding customer id in the Order database, as shown in below diagram.

Microservices with dedicated database
Order Microservice
Customer Microservice
OrderDTO

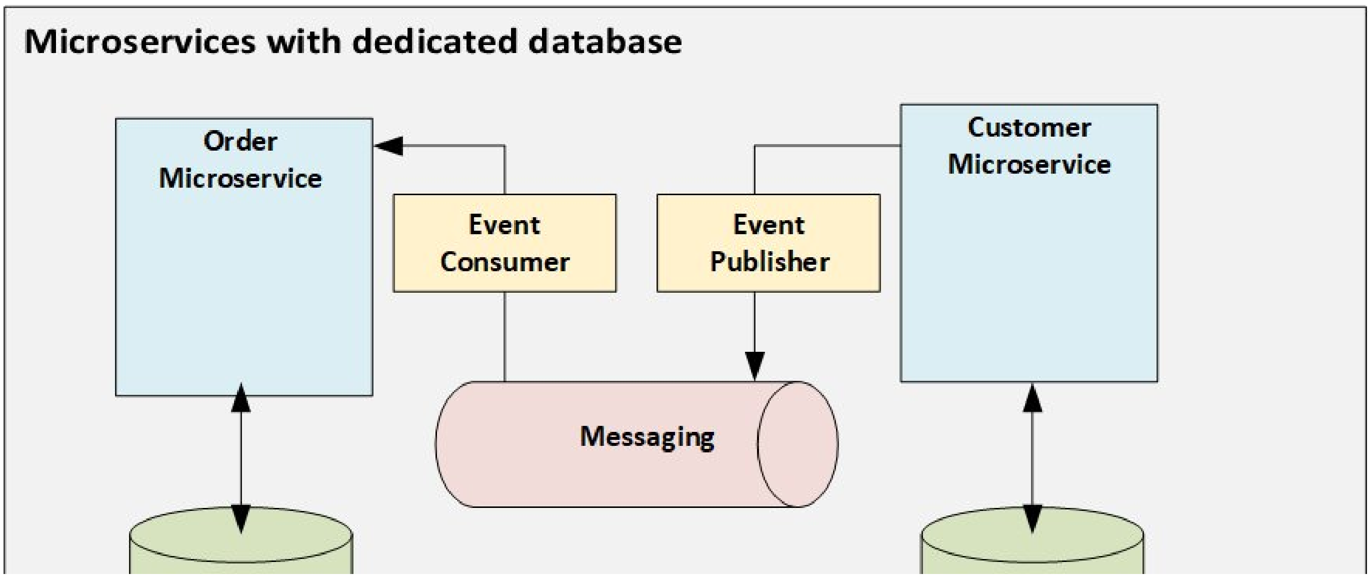


The limitation of this approach is that transaction management cannot be properly handled. If customer data is deleted, the corresponding order also has to be deleted for that customer.

Though this can be achieved with workarounds, like calling a delete service in the Order service, atomicity is not achievable in a straight forward way. This needs to be handled with customization.

To overcome this limitation, we can integrate an event-driven architecture with our microservices components.

As per the below diagram, any change in the customer data will be published as an event to the messaging system, so that the event consumer consumes the data and updates the order data for the given customer changed event.



The limitation of this approach is the atomic updates between the database and publish events to the message queue cannot be handled easily. Though these types of transactions can be handled by distributed transaction management, this is not recommended in a microservices approach, as there might not be support for XA transactions in all scenarios.

To avoid these limitations, event-sourcing can be introduced in this microservices architecture.

In event-sourcing, any event triggered will be stored in an event store. There is no update or delete operations on the data, and every event generated will be stored as a record in the database. If there is a failure in the transaction, the failure event is added as a record in the database. Each record entry will be an atomic operation.

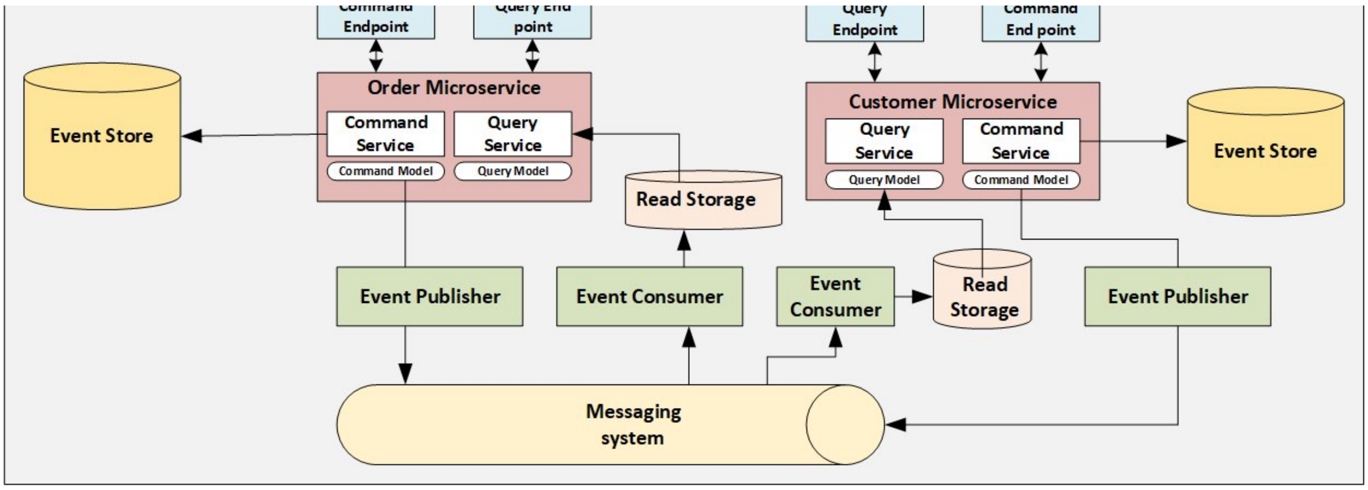
The advantages of event-sourcing are as follows:

- Solves atomicity issues.
- Maintains history and audit of records.
- Can be integrated with data analytics as historical records are maintained.

There are a few limitations, which are:

- Queries on the latest data or a particular piece of data in the event store involve complex handlings.
- To make the data eventually consistent, this involves asynchronous operations because the data flow integrates with messaging systems.
- The model that involves inserting and querying the data is the same and might lead to complexity in the model for mapping with the event store.
- The event store capacity has to be larger in storing all the history of records.

Now we integrate CQRS (Command Query Responsibility Segregation) with event sourcing to overcome the above limitations.



CQRS is another design pattern used in microservices architecture which will have a separate service, model, and database for insert operations in the database. This acts as a command layer and separate service, model, and database for query data that acts as a query layer.

The read database can store a denormalized model where databases like NoSQL (that are horizontally scalable) can be leveraged.

The command layer is used for inserting data into a data store. The query layer is used for querying data from the data store.

In the Customer microservice, when used as a command model, any event change in customer data, like a customer name being added or a customer address being updated, will generate events and publish to the messaging queue. This will also log events in the database in parallel.

Th event published in the message queue will be consumed by the event consumer and update the data in the read storage.

The Customer microservice, when used as a query model, needs to retrieve customer data that invokes a query service, which gets data from read storage.

Similarly, events published across microservices also have to be passed through a message queue.

The advantages of CQRS integrated with event sourcing and microservices are:

- Leveraging microservices for modularity with separate databases.
- Leveraging event sourcing for handling atomic operations.
- Maintain historical/audit data for analytics with the implementation of event sourcing.

- CQRS having separate models and services for read and insert operations.
- Request load can be distributed between read and insert operations.
- Read operations can be faster as the load is distributed between read and insert services.
- Read model or DTO need not have all the fields as a command model, and a read model can have required fields by the client view which can save the capacity of the read store.

The limitations of this approach are:

- Additonal maintenance of infrastructure, like having separate databases for command and query requests.
- Models should be designed in an optimal way, or this will lead to complexity in handling and troubleshooting.

Topics: MICROSERVICE BEST PRACTICES, MICROSERVICES, EVENT-DRIVEN ARCHITECTURE, CQRS, EVENT SOURCING

Popular on DZone

- Ping-Pong Implementation: JSR-356
- OutOfMemoryError: Kill Process or Sacrifice Child – Causes and Solutions
- One-Step Ingress - Use One Command To Secure APIs and Microservices
- Best Tips and Tricks on How to Work Across Time Zones

Microservices Partner Resources