The data management patterns for micro services.

**Database Per Service**

In this pattern, each microservice manages its own data. What implies is that no other microservice can access that data directly. Communication or exchange of data can only happen using a set of well-defined APIs.

Applications usually are not so well demarcated and the microservices need data from each other for implementing their logic. This leads to spaghetti-like interactions between various services in the application.

The success of this pattern hinges on effectively defining the bounded contexts in the application.

For a new application or system, it is easier to do so.

But for large and existing monolithic systems, it is troublesome.

Other challenges include implementing business transactions that span several microservices. Another challenge could be implementing queries that want to expose data from two or three different bounded contexts.

If done properly, the major advantages of this pattern are loose coupling between microservices. You can save your application from **impact-analysis** hell.

Also, you could scale up microservices individually. It can provide freedom to the developers to choose a specific database solution for a given microservice.

## Shared Database

Shared database could be a viable option if the challenges surrounding Database Per Service become too tough to handle for the team.

It solves the data management by adopting a much more lenient approach by using a shared database accessed by multiple microservices.

Mostly, this is a safer pattern for developers as they are able to work in existing ways. Familiar ACID transactions are used to enforce consistency.

**However, this approach takes away most of the benefits of microservices**. Developers across teams need to co-ordinate for schema changes to tables.

There could also be **run-time conflicts** when multiple services are trying to access the same database resources.

**Overall, this approach can do more harm than good in the long run.**

## Saga Pattern

Saga pattern is the solution to implementing business transactions spanning multiple microservices.

A **Saga** is basically a sequence of local transactions. For every transaction performed within a Saga, the service performing the transaction publishes an event. The subsequent transaction is triggered based on the output of the previous transaction. And if one of the transactions in this chain fails, the Saga executes a series of compensating transactions to undo the impact of all the previous transactions.

To understand this better, let’s take a simple example. Assume that there is a food-delivery app. When a customer tries to order food, below steps can occur:

* Food Order service creates an *order.* At this point, the order is in PENDING state. A Saga manages the chain of events.
* The Saga contacts the restaurant via the Restaurant service.
* The Restaurant service attempts to place the order with the chosen restaurant. After getting a confirmation, it sends back a reply.
* The Saga receives the reply. And depending on the reply, it can Approve the order or Reject the order.
* The Food Order service then changes the state of the order. If the order was Approved, it would inform the customer with the next details. If Rejected, it will also inform the customer with an apology message.

This is drastically different from the usual point-to-point call approach. This approach adds complexity.

## API Composition

This pattern is a direct solution to the problem of implementing complex queries in a microservices architecture.

In this pattern, an API Composer invokes other microservices in the required order. And after fetching the results it performs an in-memory join of the data before providing it to the consumer.

As evident, the downside to this pattern is the use of inefficient in-memory joins on potentially large datasets.

## Event Sourcing

## Event sourcing mainly tries to solve the problem of atomically updating the database and publishing an event.

In event sourcing, you store the state of the entity or the aggregate as a sequence of state changing events. A new event is created whenever there is an update or an insert. The event store is used to store the events.

You can use this pattern in conjunction with CQRS. By doing so, a lot of challenges around event handling and maintaining query data can be solved.

However, as a drawback, this pattern imposes an unfamiliar programming style. Also, the data is eventually consistent which may not be suitable for some use cases.

## CQRS

CQRS or Command Query Responsibility Segregation is an attempt to get around the issues with API Composition pattern.

An application listens to domain events from other microservices and updates the view or query database. You can serve complex aggregation queries from this database. You could optimize the performance and scale up the query microservices accordingly.

The downside to this is increase in complexity. All of a sudden, your microservice should be handling events. This can cause latency issues where the view database is eventually consistent rather than always consistent. It can also increase code duplication.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_