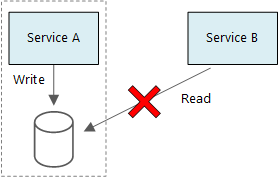
Data Considerations in Microservices

An underlying principle of microservices is that each service is responsible for its own data. What this means is that if Service A owns Data Store A, Service B cannot access Data Store A directly. It must access it through Service A.



Isolating the data store this way avoids unintentional coupling between services, preserve the ability of independent deployment and enables each service to choose the best data store and implementation for the problems it is trying to solve. Despite these benefits, there are also challenges

# Challenges

Each service can store the same data but optimized for their own purpose (i.e. transaction, analytics, reporting, archiving, etc.). This redundancy can lead to issues with data integrity and consistency. Therefore, you need to consider how to propagate updates across all services and handle case of eventual consistency.

## Guidelines to Managing Data

Isolate ACID transactions: There are places in your application where eventual consistency is acceptable and where strong consistency is required.

Expose data through an API: There should be a single source of truth for your data. That information should be exposed through an API. Other services may hold the same information but since they can be out of date, they should not be relied upon as the source of truth.

Use Scheduler Agent Supervisor and Compensating Transaction pattern to keep data consistent across several services.

Store only what the services needs: Since a service is responsible for its own data, it should only store what it needs. This limits its responsibilities and introduce strong cohesion. Furthermore, this allows different storage technology to be considered. For instance, in a delivery system, we might be interested in holding the status of current delivery and those of historical ones. For current delivery, we expected a great number of read/write since customer will often requests an update on the status and the status might change a lot until reception. However, once the package is delivered the information remains static and are seldom requested. We have two distinct type of storage and an example of a transition of source of truth.

Avoid chatty services: If two services are constantly communicating with each other’s, consider redrawing the boundaries. These services might be better off merged.

Use event driven architecture style: Services would publish events and interested service would subscribe to them in order to obtain information. This is much more favorable than polling to reduce communication. Nevertheless, this could still be a bottleneck so consider batching or aggregation of events.

# References

<https://docs.microsoft.com/en-us/azure/architecture/microservices/data-considerations>