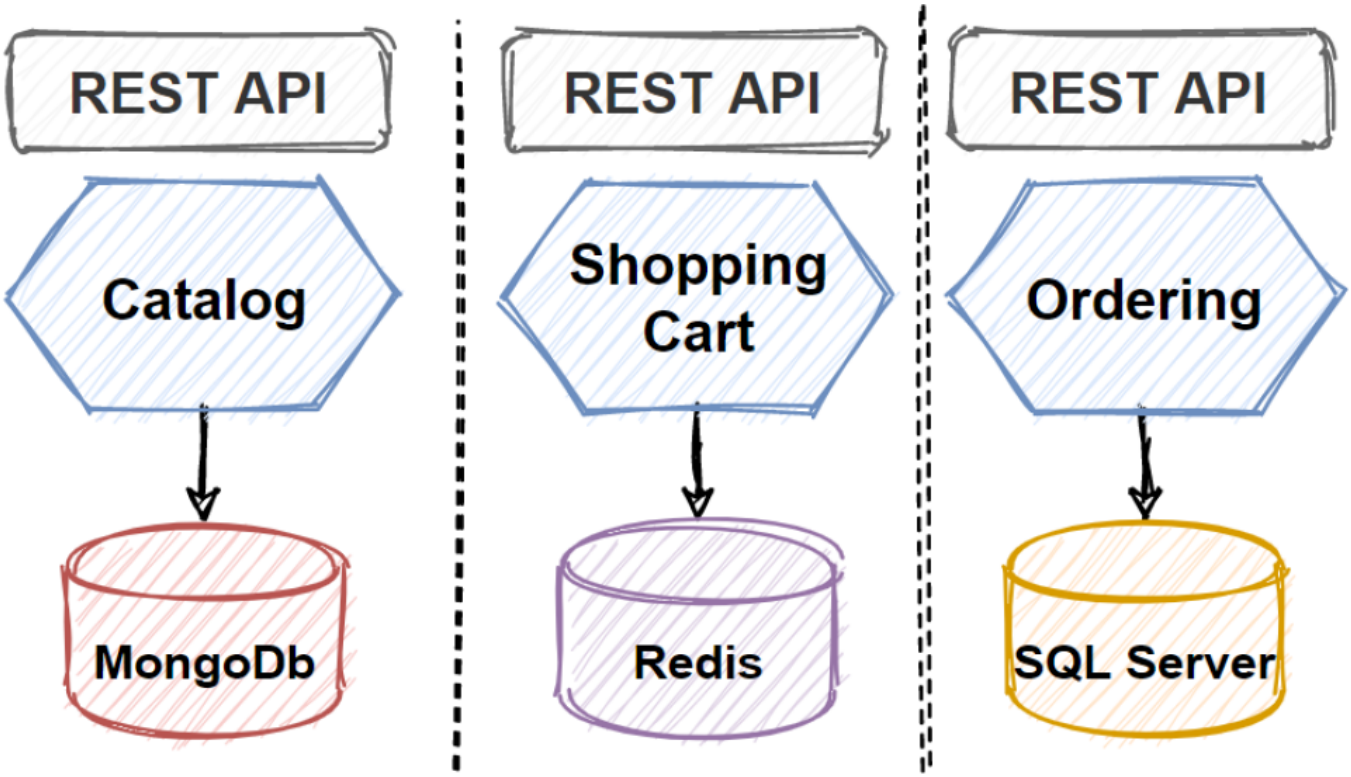
Microservices Database Management Patterns and Principles

Mehmet Özkaya Follow
Sep 8, 2021 · 7 min read

In this article, we are going to discuss **Microservices Data Management** in order to understand **data considerations** for **microservices**. As you know that we learned **practices** and **patterns** about **Microservices Data Design patterns** and add them into our **design toolbox**. And we will use these **pattern** and **practices** when **designing e-commerce microservice architecture**.



By the end of the article, you will learn how to **manage data** in **Microservices Architectures** with applying **Microservices Data Design patterns** and **principles.**

Step by Step Design Architectures w/ Course

I have just published a new course — Design Microservices Architecture with Patterns & Principles.

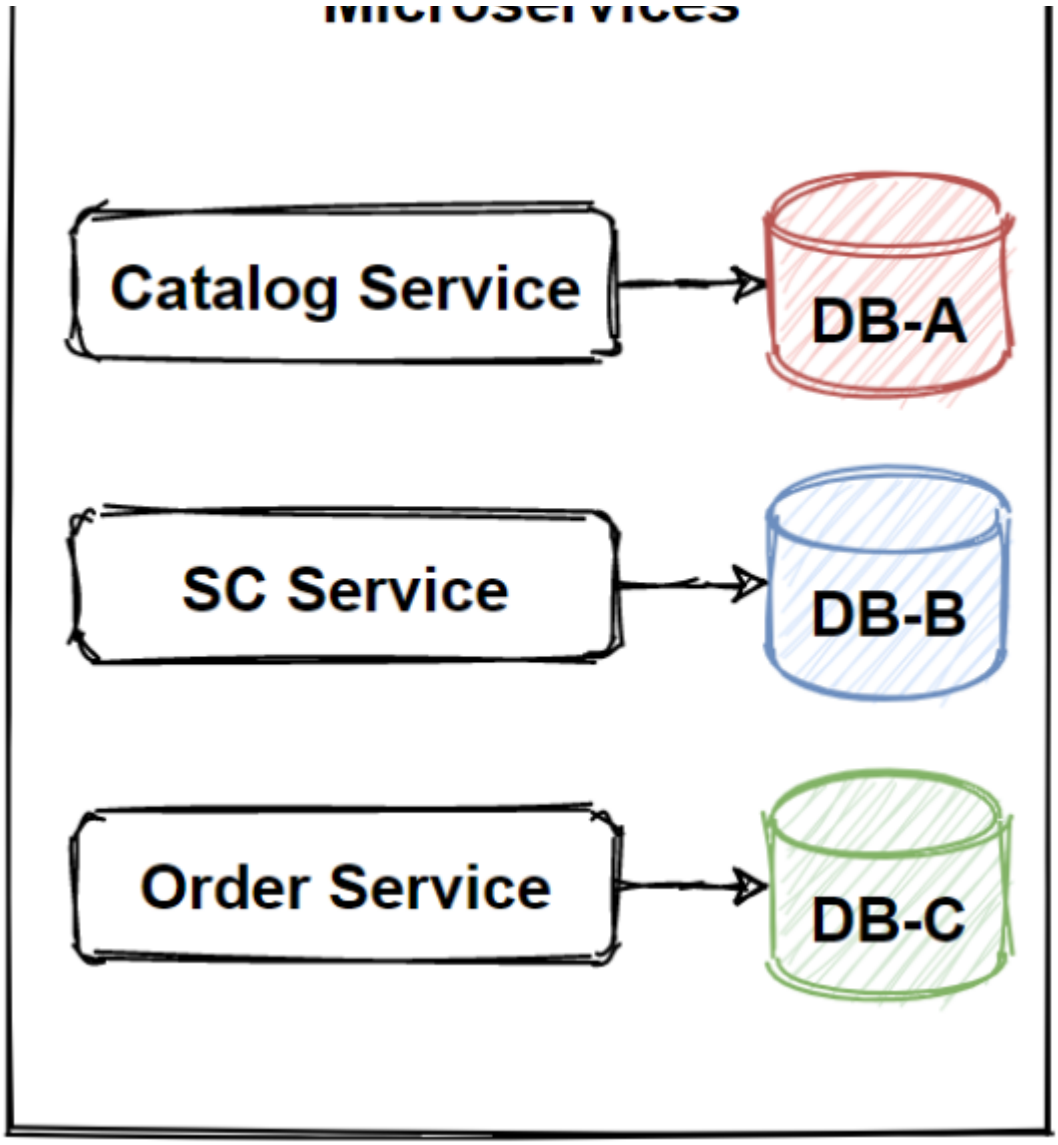
In this course, we're going to learn **how to Design Microservices Architecture** with using **Design Patterns, Principles** and the **Best Practices.** We will start with designing **Monolithic** to **Event-Driven Microservices** step by step and together using the right architecture design patterns and techniques.

Microservices Database Management Patterns and Principles

We are going to talk about **Microservices Database Management Patterns** and Principles. In which tools, which **patterns**, which **principles**, which **best practices** we can use when considering data management in microservices ?

We already said that managing a microservices database is huge challenging job. And also we said that we should have a **strategy**. So how we can set a strategy ? Of course using **best practices** and **pattern** on this area.

So we should learn about patterns that solve the issues about microservices data **decentralization**. Microservices should have **own data** and microservices need to **interact** and **share data** with each other. When interact or **sharing data** between **microservices**, The problem is, you can't use **ACID transactions** between distributed systems. That means its challenging to **implement queries** and transactions that visits to several microservices.



I will give only the captions of patterns, **principles** and **best practices** for **Microservices Database Management**, and after this article we will elaborate this patterns and principles.

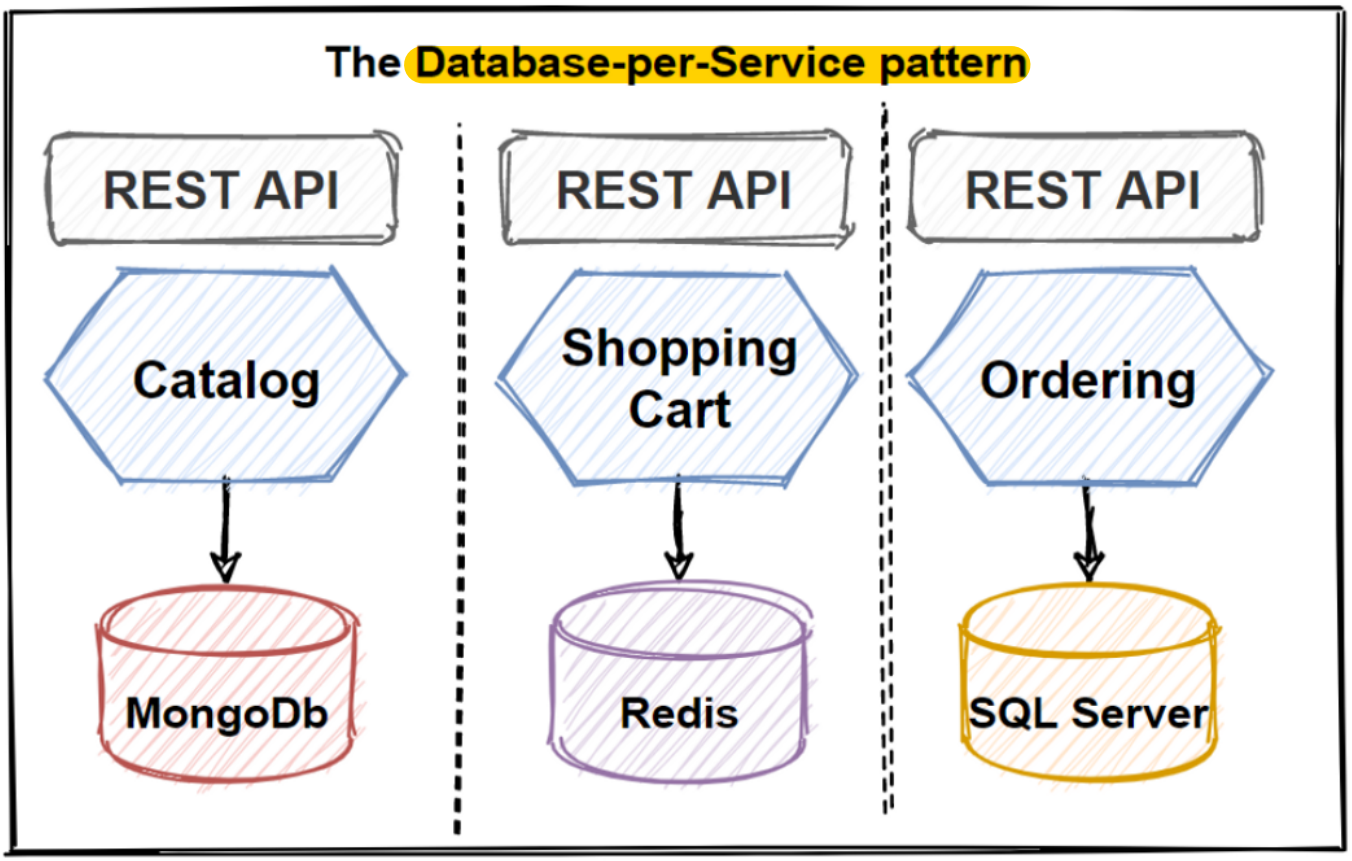
Let's look at the collection of patterns related to microservices data management. We have **5 common data-related patterns** and **1 anti-pattern;**

- The Database-per-Service pattern
- The API Composition pattern
- The CQRS pattern
- The Event Sourcing pattern

- The Saga pattern
- The Shared Database anti-pattern

The Database-per-Service Pattern

This is the one of the main characteristic of the microservices architecture. In order to be a **loose coupling** of services, each microservice should have its own private database. So when **designing database architecture** for microservices, it will almost always requires the **database-per-service pattern.**



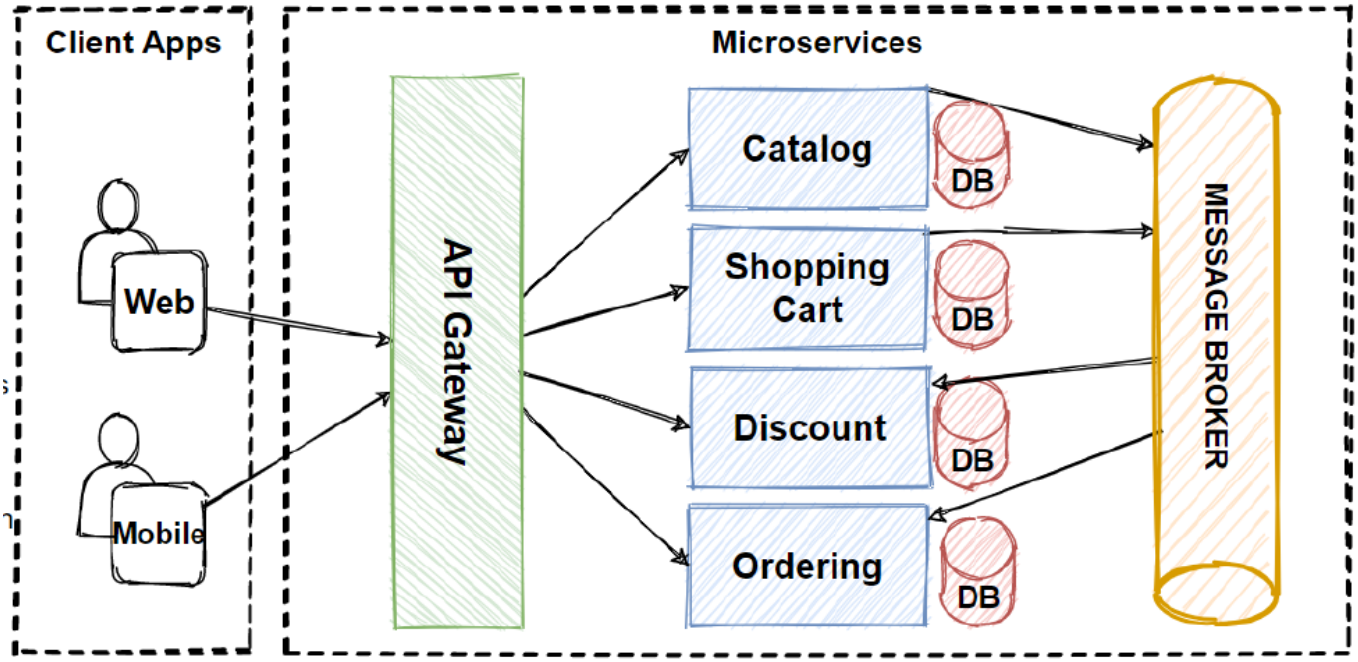
When we are shifting to the **monolithic architecture** to **microservices architecture**, one of the first things to do is decomposes databases.

So we should **decomposes database** into a **distributed data model** with many smaller databases for particular microservice. This will become to design a database per microservice. The **database per microservice** provides many benefits, we can say that it provide to evolve rapidly and easy to scale applications.

Actually the main benefit of **database per microservices** is Data schema changes can perform without any impact on other microservices. So that means if any data failures happened, it wont be affect other microservices.

And **scaling independently** is also very powerful because the volume of request can come to microservices differently, if 1 microservices peek the requests that only that microservice can **scale independently**.

Separating databases can gives us to abilities to pick the best optimized database for our microservices. We can **choices** include **relational**, **document**, **key-value,** and even **graph-based data stores**.

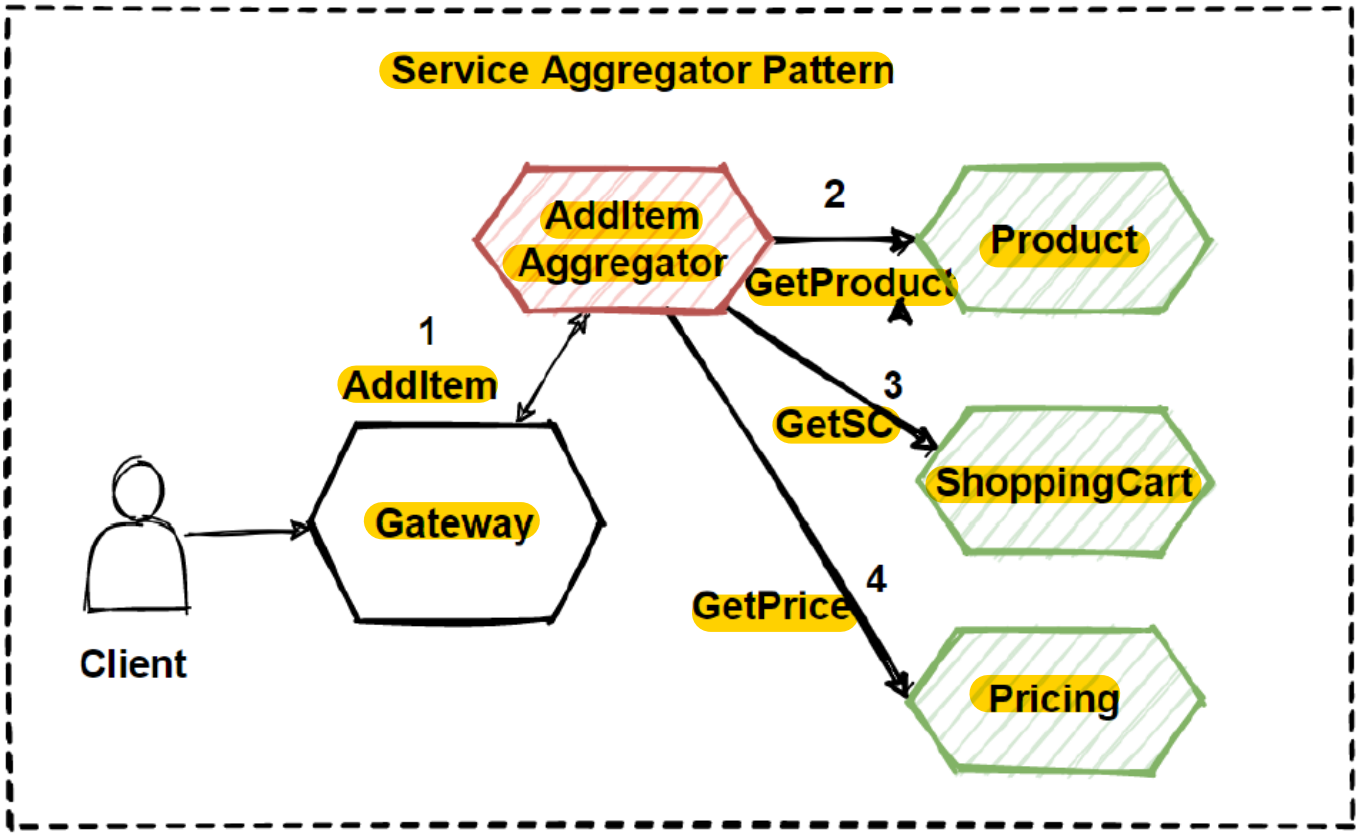


If look at our architecture each microservice supports a different type of databases. The **product catalog microservice** uses a **no-sql document database** which is **mongodb**. The **shopping cart** microservice uses a distributed cache that supports its simple, **key-value data store**. The ordering microservice uses a relational database to accommodate the **rich relational structural data**. So this segregation gives us to use power of databases in right place and able to **scale independently** according to load of the microservices.

The API Composition Patterns

In distributed microservices, retrieving data from several services also need a set of patterns and practices. As we already seen **API Gateway patterns** and practices, this is good place to remember when thinking how to handle queries in microservices.
These patterns are;

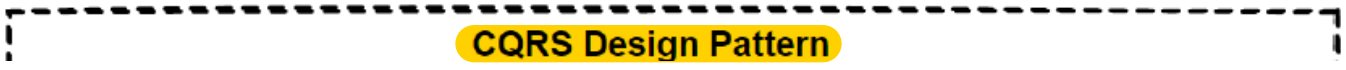
API Gateway Pattern
Gateway Routing pattern
Gateway Aggregation pattern
Gateway Offloading pattern

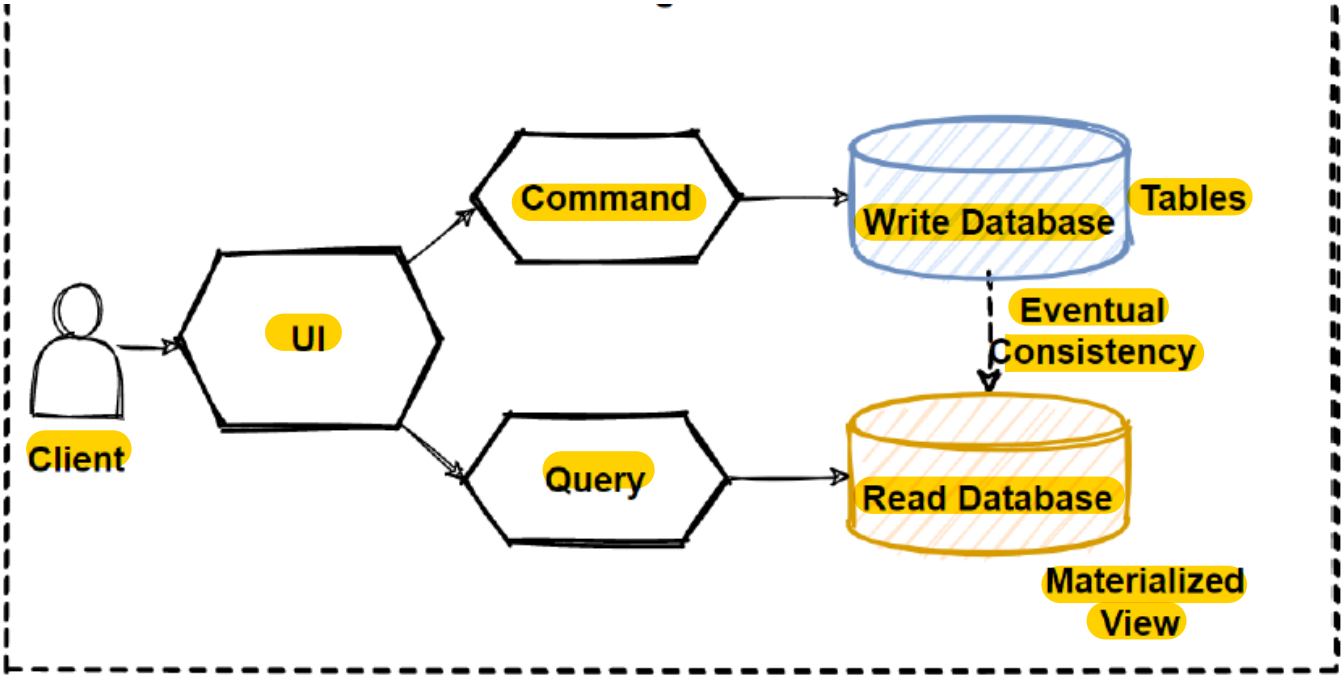


So when implements a query by invoking several microservices, we will follow the API Composition, Gateway Aggregation patterns for combining the results.

The CQRS Pattern

The **Command Query Responsibility Segregation (CQRS)** is provide to separate commands and queries database in order to better perform querying several microservices.

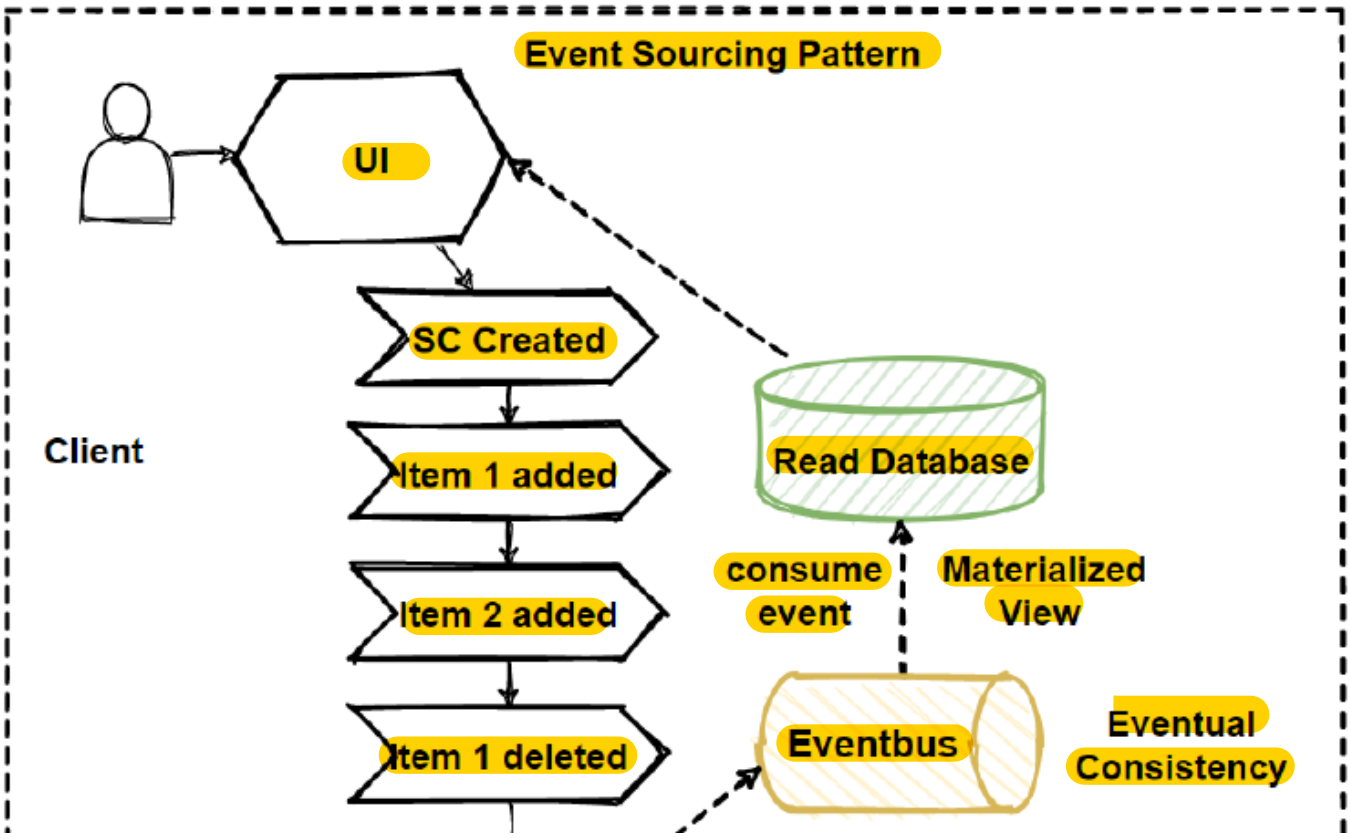


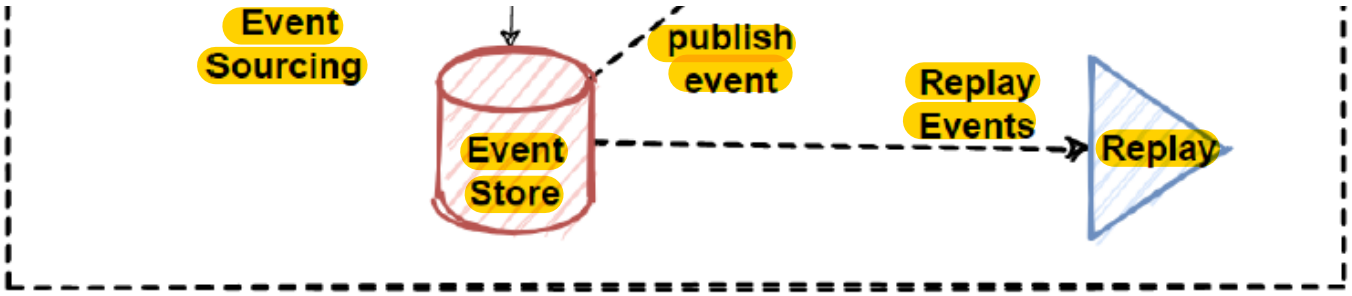


Its based on write-less, read-more approaches, if we have this kind of operation behaviors its good to use this pattern.

The Event Sourcing Pattern

The Event Sourcing pattern basically provide to accumulate events and **aggregates** them into **sequence** of **events** in databases.

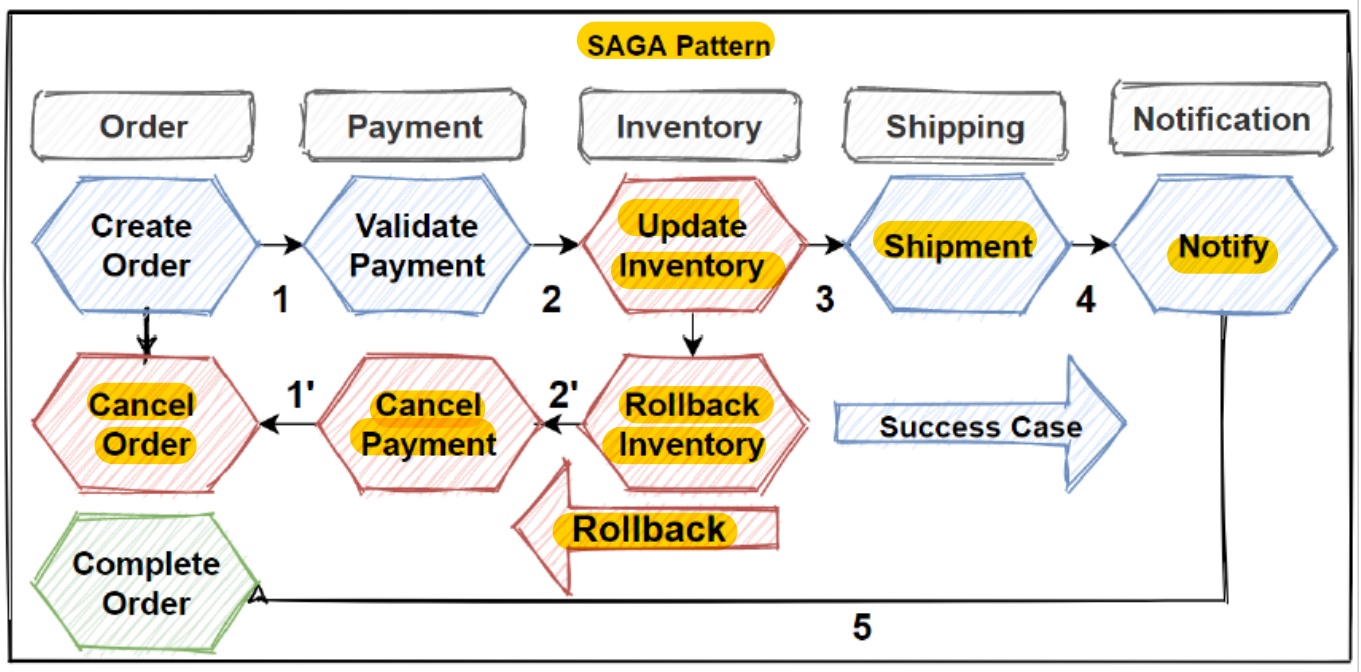




By this way we can replay at certain point of events. This pattern is very well using with cqrs and saga patterns.

The Saga Pattern

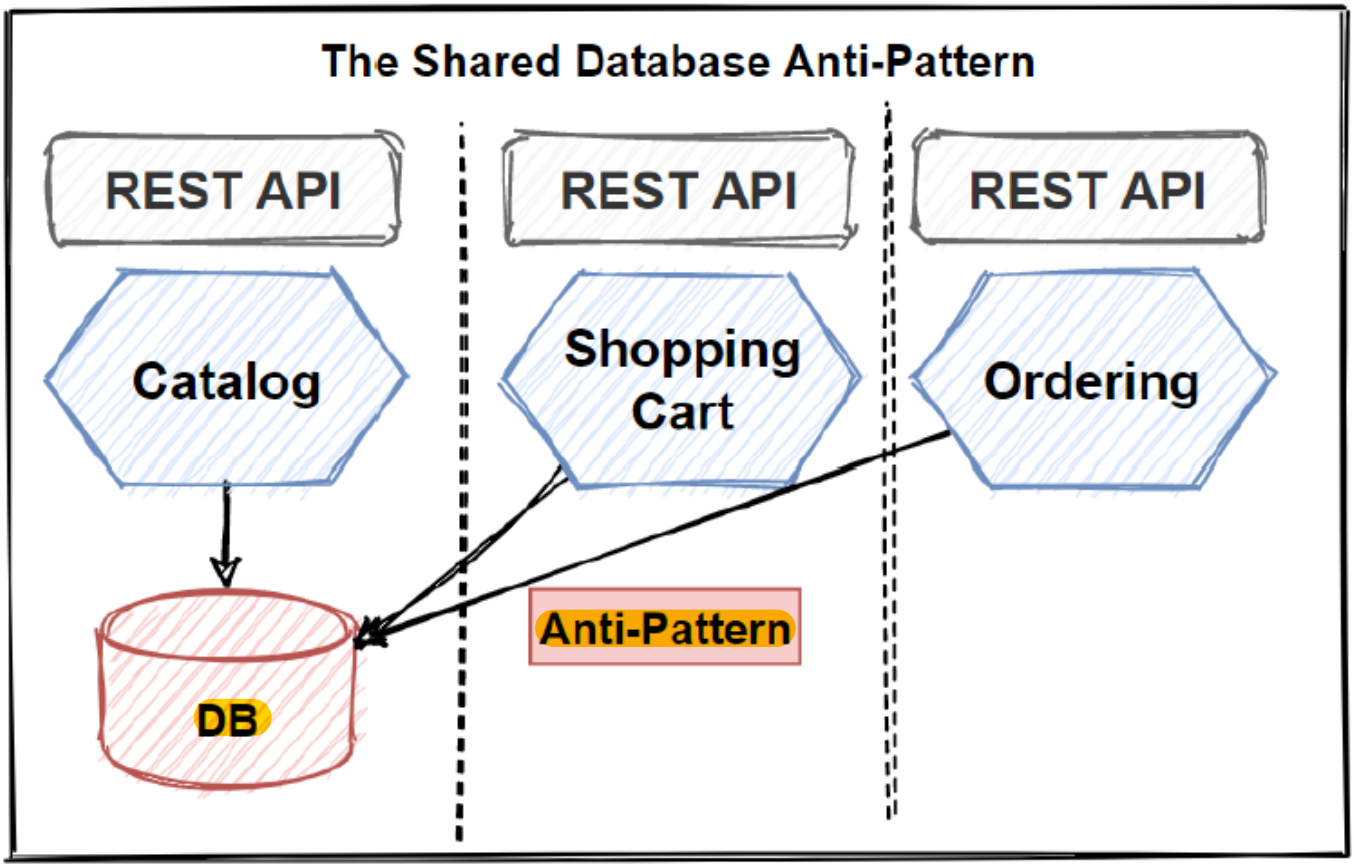
Transaction management in really hard when it comes to microservices architectures. So in order to **implementing transactions** between several microservices and **maintaining data consistency**, we should follow the **SAGA pattern**. Saga pattern has two different approaches:



Choreography — when exchanging events without points of control
Orchestration — when you have centralized controllers

The Shared Database Anti-Pattern

If you don't follow The Database-per-Service pattern and use Shared Database for several microservices, then it is anti-pattern and you should avoid this approaches.



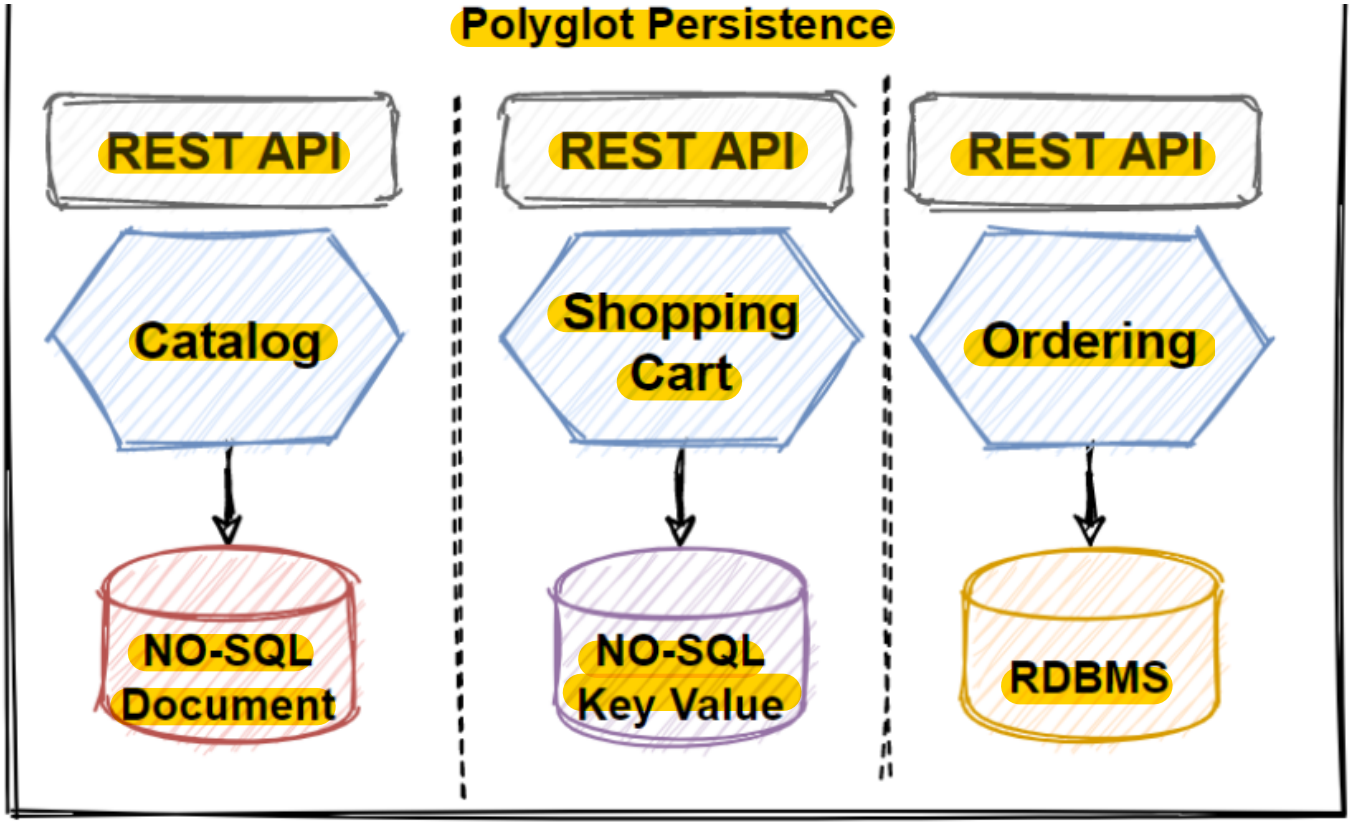
You can create a single shared database with each service accessing data using local **ACID transactions**. But it is against to microservices nature and will cause serious problems in the future of applications. At the end of the day, you will face to develop big a few monolithic applications instead of microservices.

So as you can see that we have seen several patterns and practices for managing data in microservices now lets elaborate this principles. So with a shared database, we will **loosing power** of **microservices** like loose coupling and services independency. Also shared database can block microservices due to **single-point-of-failure**.

In order to get benefits of microservices best features we should follow the database-per-service pattern.

Polyglot Persistence

The microservices architecture enables using different kinds of data storing technologies for different services aka applying **polyglot persistence**. Each development team can choose the persistence technology that suits the needs of their service best.



Martin Fowler has great article about **Polyglot Persistence** principle and explains that polyglot persistence will come with a cost — but it will come because the benefits are worth it.

When **relational databases** are **used inappropriately**, they give damaged on application development. So we should understand how usage is required for microservice, For example If only **looked up page elements by ID**, and if you had no need for transactions, and no need to share their database, then its not meaningful to use relational database.

A problem like this is much better suited to a key-value no-sql databases than the corporate relational databases.

So this things comes to create a question;

- **How to Choose a Database for Microservices ?**

Step by Step Design Architectures w/ Course

I have just published a new course — Design Microservices Architecture with Patterns & Principles.

In this course, we're going to learn **how to Design Microservices Architecture** with using **Design Patterns, Principles** and the **Best Practices.** We will start with designing **Monolithic** to **Event-Driven Microservices** step by step and together using the right architecture design patterns and techniques.

Microservices Data Management Design Patterns Microservice Architecture

Microservices Pattern