ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/azra-jabeen-mohamed-ali/376

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 2 & Iss: 1

Research Article

Exploring Database design patterns of Microservices

Azra Jabeen Mohamed Ali*

Citation: Ali AJM. Exploring Database design patterns of Microservices. J Artif Intell *Mach Learn & Data Sci 2024*, 2(1), 1732-1735. DOI: doi.org/10.51219/JAIMLD/azra-jabeen-mohamed-ali/376

Received: 02 January, 2024; **Accepted:** 18 January, 2024; **Published:** 20 January, 2024

***Corresponding author:** Azra Jabeen Mohamed Ali, Independent researcher, California, USA, E-mail: Azra.jbn@gmail.com

Copyright: © 2024 Ali AJM., Postman for API Testing: A Comprehensive Guide for QA Testers., This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

A B S T R A C T

This paper discusses the thorough exploration of the Database design patterns associated with the Microservices. Microservices have transformed the software development sector by encouraging modularity, scalability and maintainability, which enables businesses to react to shifting consumer needs and technology breakthroughs faster. The study's main research question explores the careful consideration of database design in microservice architecture due to its distributed nature of microservices. It also provides a thorough analysis of several microservice's Database patterns and the way it handles its own data for improved scalability, flexibility and isolation. This paper is therefore meant to be more development-environment centered and infrastructure agnostic. Developers and architects who wish to concentrate on code, patterns and implementation specifics will find this part most interesting.

Keywords: Micro Services, Design patterns, Event Sourcing, Api Composition pattern , Database design patten, monolithic

1. Introduction

1.1. Microservice Architecture:

Microservice architecture is a design methodology that divides a large application into smaller, autonomous services, each of which focuses on a distinct business function. Therefore, the back end is the main focus of this method, even though the front end can also use a microservices design. Each service runs independently and communicates with other processes using protocols including HTTP/HTTPS, WebSocket's and AMQP.

Business-critical enterprise applications need to provide updates fast, frequently and reliably in order to thrive in today's unstable, uncertain, complex and ambiguous reality. As a result, corporations are divided into small, cross-functional teams with limited connections. Each team uses DevOps methodologies to deploy software. Specifically, it makes use of continuous deployment. An automated deployment pipeline tests the team's stream of frequent, small modifications before they are put into production. The intention is to allow developers to leverage microservices to speed up application releases by allowing teams to deploy each microservice as needed.

1.2. Why are Microservices Architectures used by Businesses?

Most firms start by constructing their infrastructures as a collection of closely related monolithic applications or as a single monolith. The monolith does a number of things. All of the programming for those functionalities is included in a single, cohesive piece of application code. Because the code for these functions is so intertwined, it is difficult to understand. The code of an entire program may break as a result of a single feature addition or alteration in a monolith. This makes any change, no matter how simple, expensive and time-consuming. As upgrades are done, programming becomes more complicated until scaling and upgrading are practically impossible.

Businesses can no longer make additional changes to their coding over time without starting over. Businesses may find

themselves stuck with antiquated procedures for a long time after they should have modernized, as the process soon becomes too difficult to handle.

In addition to other pertinent factors, company objectives will determine which pattern (or patterns) is best to use. For microservices, there are numerous design patterns, each with its own advantages and disadvantages. Design patterns are grouped according to their intended use like Decompose patterns, Observability patterns, Integration patterns, Database patterns, Cross-Cutting concern patterns. Database design patterns, including Database per Service, Shared Database per Service, CQRS, Event Sourcing and Saga patterns, are the main topic of this article.

1.3. Database Per Service: The majority of services require data to be stored in a database. Let's picturize that we are developing a ott platform application like Hulu, Disney,HBO which has Subscription Service and Customer Service. Customer Service stores information about cutomers whose data is stored in relational database and subscription service stores information about the subscriptions whose data is stored in NoSQL database (Figure-1).

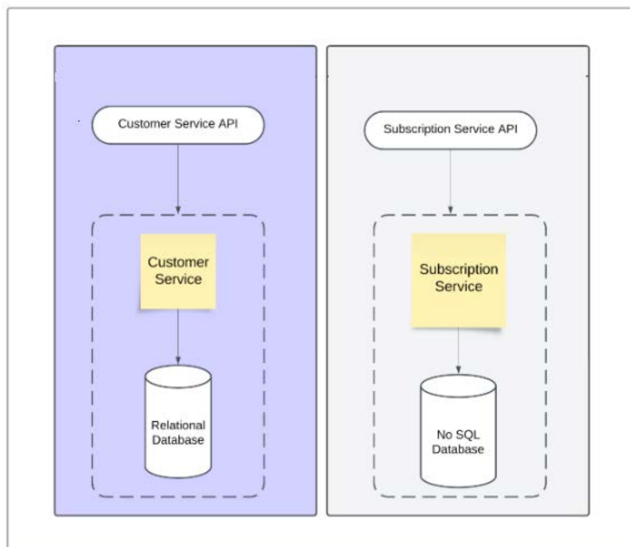

Figure-1

In this design pattern every microservice ought to have its own database that it controls on its own. It cannot be accessed by other services directly. Instead, the persistent data of each microservice exclusive to that service and only available through its API. By doing this, microservices are guaranteed to be loosely connected and are free to develop their own data models and storage systems. Various microservices can employ various database technologies according to the use case (e.g., NoSQL for rapid reads and good at storing complex and unstructured data, SQL for transactional). Only a service's database is used in transactions. Independent scalability is possible for each service and its database.

If the database is relational, there is no need to set up a database server for every service. The choices for such a scenario are

1.3. Private-tables-per-service: Each service has a set of tables that are private to that service and can only be accessed by that service.

Schema-per-service - Every service has its own database schema, which is known as schema-per-service.

1.4. Database-server-per-service: Every service has a database server of its own.

For the purpose of establishing a barrier and preventing it from utilizing other service tables, each microservice should have its own database ID.

The following issues need to be taken into consideration while designing the database architecture for microservices: It is necessary to loosely couple services. They are independent in terms of development, deployment and scaling. Multiple-service invariants may be enforced via business transactions. Data owned by various providers may need to be queried in certain business transactions. Sometimes sharding and replication are necessary for databases to grow. The amount of data storage needed varies per service.

1.5. Challenges: Data duplication: It's possible for certain data to be repeated between services. Data consistency: It gets more difficult to maintain consistency between services. Managing transactions that cross several databases or services is known as distributed transactions.

1.6. Shared Database per Service:

Microservices work best when there is just one database per service, which is only feasible in greenfield applications (new development, no prior work done that poses constraints on application) created with DDD (Domain Driven Design). Denormalization is difficult when attempting to convert a monolithic program into microservices. Shared database per Service is best suited for brownfield applications, this is a fantastic place to start when trying to break the application into more manageable chunks.

Single database is shared by multiple services. Through local ACID transactions, each service has unrestricted access to data that belongs to other services.

Let's consider that we are developing a OTT platform application like Hulu, Disney, HBO which has Subscription Service and Customer Service. Customer Service stores information about cutomers whose data is stored in the Customer table in relational database and subscription service stores information about the subscriptions in Subscription table in the same relational database. With the help of Shared Database per Service pattern, Customer Service and Subscription Service can access each other's table **(Figure-2)**.

1.7. Pros: Easy to manage a single database. Very familiar and straightforward ACID transactions can be used to enforce data consistency.

1.8. Challenges:

- **Runtime Coupling:** Availability of one service is impacted by the availability of another service. The Subscription Service will be blocked if a Customer Service transaction locks the subscription table.
- A single database might not be able to meet all the services' needs for data access and storage.
- **Development time coupling:** Developers of other services that utilize the same tables must coordinate schema changes with those working on the Subscription Service, for

instance. Further coordination and this linkage will slow down progress.

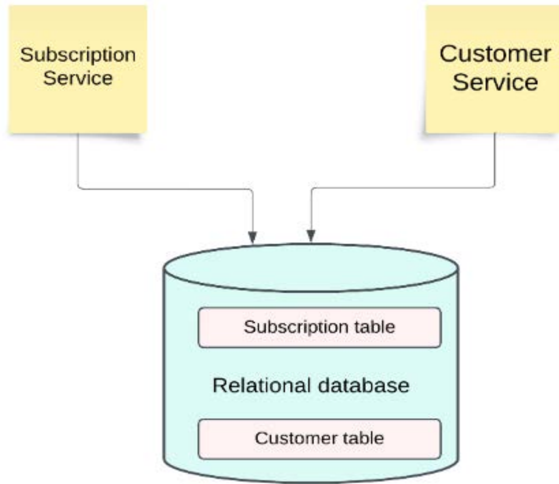

Figure-2

2. API Composition Pattern

The necessity for this architecture is created by the Database per Service pattern. Services may utilize APIs to retrieve data from other services rather than sharing a database. In this approach, a composite service calls the APIs of several microservices to aggregate data from them, then composes the answer to produce a single, cohesive outcome. In order to execute a query, an API Composer has to be defined that calls the data's services and joins the results in-memory **(Figure-3)**.

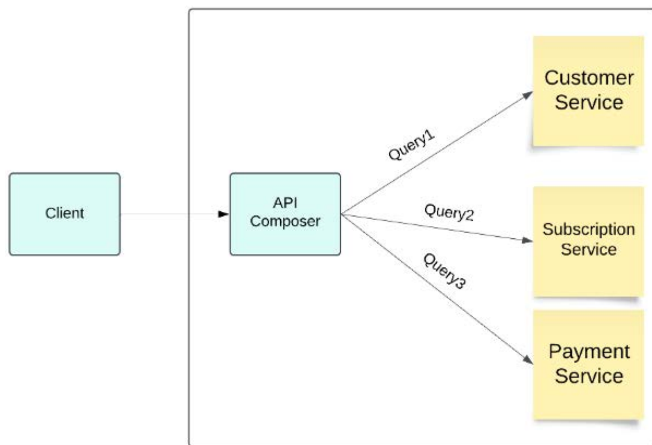

Figure-3

API Composition Pattern depicts a "client" submitting a request to a "API Gateway," which then makes calls to several "Microservices" to obtain data. An "API Composer" component is in charge of integrating the output from these services into a single response, thereby carrying out an in-memory join to provide the client with a single data set.

2.1. Pros: In a microservice design, it's an easy method of querying data. Instead of using shared data stores, each service is in charge of its own data. Data from many services can be combined and presented to the client by the composition layer.

2.2. Challenges: The system becomes more complex as a result of services becoming reliant on one another. Performance may have an affect especially for large datasets when making several API calls for a single query. If one service is down, the composed response might be incomplete or fail.

3. CQRS (Command Query Responsibility Segregation)

In microservice architecture, the CQRS pattern provides a way to implement a query that pulls information from several services. The CQRS pattern divides the data model into two parts: one for writing operations, which manages the command side such as CREATE, UPDATE and DELETE requests and one for read operations, which handles the query side using views. This technique makes it possible to optimize the write side for consistency and integrity and the read side for querying (using denormalized data).

3.1. Working model of CQRS: (Figure-2) The basic step is to determine the read and write activities of the application so that the read operations are segregated to read model and write operations are segregated into Command model. The command model and read model are physically isolated. The command / write operation modifies the data based on the command and does not return any data. Whereas the read operation does not modify the data and returns the data.

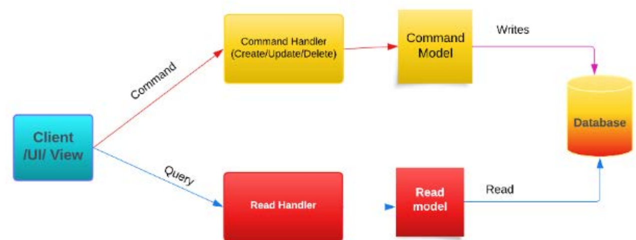

Figure-2

3.2. Working model of CQRS with Event Sourcing pattern: (Figure-3) In the same working CQRS, say suppose if the databases used for read and command operations are different (one as relational DB and other as NoSQL Db), both data should be in sync. For such synchronization process, Event handlers take the command model data and send it asynchronously to another module in order to save the same data to the materialized view (NoSQL database). All the command requests or data changes are created as an event and stored as a series of events for tracking purpose. This provides an audit trail by storing the entire history of state changes.

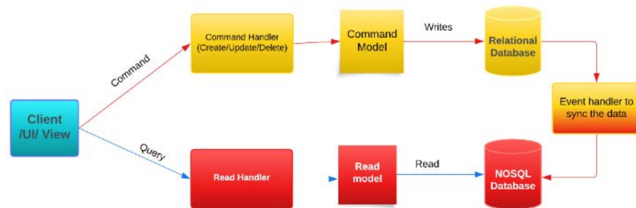

Figure-3

3.3. Pros of CQRS:

3.3.1. Read and write optimization: The command model can be optimized for consistency and write efficiency, while the query model can be optimized for quick readings. Complex queries and writes can be managed more effectively by dividing concerns and thereby performance is increased. Enables to scale the read and write sides independently.

3.3.2. Challenges:

In terms of data synchronization, it will be challenging to maintain the read and write models in sync. Consistency issues could arise when the system takes some time to update to the most recent state. It is necessary to carefully analyze data storage

and retrieval patterns in order to store all events. Rebuilding the state after an event can be challenging, particularly when there are more occurrences.

3.3.3. Saga pattern:

The Saga pattern is a way to implement cross-service transactions **(Figure-4)**. If an application has multiple microservices with different database records, then this saga pattern is one of the best choices to allow the transactions between multiple microservices with rollback functionalities.

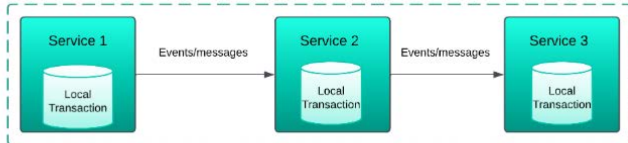

Figure-4

A series of local interactions is called a saga. Every local transaction in the saga causes a message or event to be published, updating the database and setting off the subsequent local transaction. In the event that a local transaction fails due to a violation of a business rule, the saga initiates a sequence of compensatory transactions that reverse the modifications made by the previous local transactions. This can be achieved by two ways namely Choreography orchestration.

3.3.4. Choreography:

As the name mentioned above, it is based on choreography-based saga meaning the sequential steps to be carried out. Every local transaction trigger local transaction in other services by publishing domain events. There will be no centralized coordination. After completing a transaction, a service will publish an event using the choreography approach. Other services may occasionally react to those events that are published and carry out duties in accordance with their coded instructions. Depending on defaults, these secondary tasks may or may not also post events.

3.3.5. Orchestration:

Similar to orchestration orchestrator coordinates with the services and decides what local transactions to be executed. An orchestration technique will use an object to orchestrate transactions and publish events, which will cause other services to finish their tasks in response.

3.3.6. Pros:

The pattern's distributed design makes it possible to handle long running process more effectively. To reduce complexity, each microservice manages its own local transaction.

3.3.7. Challenges:

State management, compensation actions and service coordination are necessary for saga implementation which makes it bit complex. The system may only become consistent over time, similar to event sourcing, necessitating cautious management of errors and retries.

4. Benefits of Database Design Patterns in Microservice Architecture?

4.1. Decentralized Data Ownership: Each microservice usually oversees its own database in a microservices architecture. This promotes service autonomy and lessens tight coupling between services by guaranteeing that the service has complete control over its data.

4.2. Flexibility: Each service can use the database type (SQL, NoSQL, graph, etc.) that best suits its requirements thanks to microservices.

4.3. Simplified Data Management: Instead of a shared database with a complicated and generic schema, each service can have its own data model that reflects its particular domain by isolating databases.

4.4. Fault tolerance: Problems with one service's database (such as outage or performance degradation) won't directly impact other services when each microservice has its own database.

4.5. Optimized Performance: To maximize performance in microservices, database design patterns like Event Sourcing and CQRS (Command Query Responsibility Segregation) might be used.

4.6. Data Consistency and Eventual Consistency: Strong consistency, which can be difficult to maintain in a microservices context, can be avoided by using patterns like Saga or Eventual Consistency, which allow microservices to achieve eventual consistency in a distributed system.

4.7. Improved Security and Access Control: Better isolation is made possible by microservices with distinct databases, which improves security.

5. Conclusion

For microservices to stay scalable, maintainable and loosely connected, database design is essential. The particular use case, including read/write patterns, consistency requirements and the intricacy of the business logic, frequently determines the best design pattern. An architecture may employ a mix of these patterns to accommodate diverse microservice requirements.

6. References

1. https://microservices.io/patterns/data/database-per-service.html
2. https://dzone.com/articles/design-patterns-for-microservices
3. https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design
4. https://microservices.io/patterns/data/shared-database.html
5. https://microservices.io/patterns/data/api-composition.html
6. https://microservices.io/patterns/data/event-sourcing.html
7. https://microservices.io/patterns/data/saga.html
8. https://www.openlegacy.com/blog/microservices-architectu-re-patterns/
9. https://microservices.io/patterns/microservices.html