Data management patterns in microservices architecture

Giovanni Jiayi Hu
Department of Mathematics
University of Padua, Italy I-35121
Email: giovannijiayi.hu@studenti.unipd.it

***Abstract*—The microservices architecture is an approach to developing an application as a set of small independent services, which have their own lightweight stack and communicate with other similar services via well-defined interfaces. In order to achieve loose coupling, a viable strategy is for each service to have its own data store. However, this kind of architecture introduces a new range of obstacles. We must, indeed, implement transactions that work across multiple distributed services. An operation that spans services must use what's known as a saga, a message-driven sequence of local transactions, to maintain data consistency. Unfortunately, sagas lack the isolation feature of traditional ACID transactions. As a result, an application must use countermeasures to prevent or reduce the impact of concurrency anomalies caused by the lack of isolation. On top of that, writing queries in a microservice architecture reveals to be likewise challenging. Implementing queries in an existing monolithic application is relatively straightforward because it has a single database, but in a microservice architecture queries often need to retrieve data that are scattered among the databases owned by multiple services. This paper provides an overview of microservices patterns which emerge when adopting independent data stores, along with some suggested practical implementations.**

I. Introduction

A microservice is as a small self-contained application that has a single responsibility, a lightweight stack, and can be deployed, scaled and tested independently [1]. Services run in their own remote processes and communicate between each other using a lightweight mechanism such as message-queuing, which is discussed later.

The central aspect of the microservice definition is the independence, the ability for a service to change without affecting any other. As such, services are indipendently deployable and scalable in addition to offering a better fault isolation than the monolithic architecture. For example a memory leak in one service will only affect that service, while the other ones will continue to handle the requests normally.

To achieve this praised independence, the database architecture must respect the following requirements:

- Services must be loosely coupled so that they can be developed, deployed and scaled independently;
- Some business transactions must enforce invariants that span across multiple services. For instance, before placing an order, you must verify that a new order will not exceed the customer's credit limit;
- Some business transactions need to query data that are owned by multiple services. For example, displaying the orders require querying the Orders service to gather the cost and the Restaurant service to know if an order has been accepted;
- Different services might have different data storage requirements. For an Orders service a relational database could be the best choice, but other services might need a SQL database with extensions for non-relational features, such as geospatial datatypes, or even a NoSQL database. A service which provides the nearest restaurants certainly requires geospatial queries.
- Databases must support replication and sharding in order to scale. Replication improves the application availability and performance whereas sharding allows the application to handle increasing data volumes;
- Databases contain no business logic, such as stored procedures. In doing so, databases for different services can more easily combined and separated with minimal code changes [2].

A standard solution to these requirements is to keep each microservice's persistent data private to that service and accessible only via its API. The transactions of a service involve only its database. Using a database per service allows the services to be loosely coupled and guarantees higher resilience. Changes to one service's database do not impact any other services. Besides, each service can use the type of database that is best suited for its needs. For example, a service that handles text searches could use ElasticSearch [3].

However, using a database per service introduces severe drawbacks which must be carefully evaluated: implementing business transactions that span across multiple services is complex and implementing queries which join data from several databases is challenging.

II. Queries

Three problems are commonly encountered when implementing queries in a microservice architecture:

1) To perform a single task, a client may have to make multiple calls to various services, which result in high latency and poor user experience. This chattiness between a client and different services can adversely impact the performance and scale of the application;
2) Retrieving data scattered across multiple services can result in expensive, inefficient in-memory joins without a proper solution;

3) The service can store the information in a database that doesn't efficiently support the required query, for instance traditional relational databases are not efficient for geospatial queries.

A. API composition pattern

The most straightforward approach is the API composition pattern [4], which implements a query operation by invoking the services that own the data and combining the results. As example, a query operation to show the details of an order might invoke three services - `Order Service`, `Delivery Service and Payment Service` - and return aggregate the results.

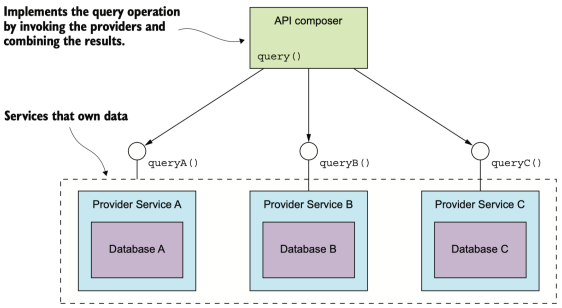

Fig. 1. The API composition pattern implements an operation by querying the service providers and combining the results. [4]

The role of API composer is usually taken by a service such as an API gateway, in which case the pattern is also called Gateway Aggregation [5]. This approach requires developing your own API gateway with a framework or library, instead of just configuring static files using an off-the-shelf API gateway such as Ambassador for Kubernetes[1].

This pattern can reduce the number of requests made to the services and improve application performance over high-latency networks, especially when caching techniques are applied.

However, the API composition pattern has several drawbacks. It can result in a very inefficient in-memory join of large datasets and it requires more computing and network resources, increasing the cost of running the application. Moreover, there is a risk that a query operation will return inconsistent data since it executes queries against multiple databases.

Lastly many queries can't be implemented using this pattern. Image an operation which retrieves a consumer's order history. In contrast to replying with the details of a single order, it returns multiple orders from different services. Requesting orders from the services by ID is only practical if those services have a bulk fetch API, which is however unlikely. Requesting orders individually will, on the other hand, be inefficient because of excessive network traffic.

On the implementation side, an API composer should call provider services in parallel in order to minimize the response time for a query operation. Sometimes, though, an API composer needs the result of one service in order to invoke another one. The logic to efficiently execute a mixture of sequential and parallel service invocations can therefore be complex.

In order for an API composer to be maintainable as well as performant, it could use a reactive design based on Reactive streams (explained later) or some other equivalent abstraction, such as Futures or Promises depending on the programming language [6]. Reactive streams and futures or promises make it possible to compose asynchronous operations or define a pipeline of operations to be invoked upon completion of the computation. This is in contrast to callback-heavy or more imperative direct blocking approaches.

Ultimately, whether you can use the API composition pattern to implement a particular query operation depends also on how the data is partitioned and the capabilities of the APIs exposed by the services that own the data.

B. CQRS pattern

In contrast to the API composition pattern, the Command query responsibility segregation (CQRS) pattern [7] is more powerful as we shall explain, but it's also more complex. It maintains one or more view databases whose sole purpose is to support queries.

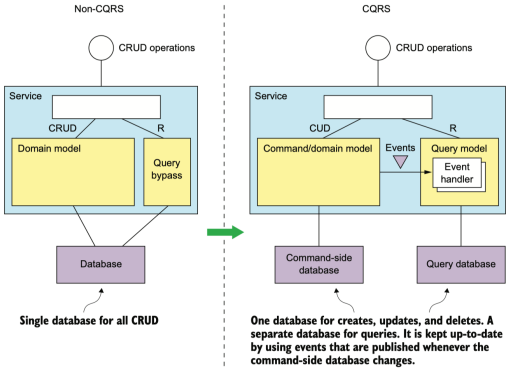

Fig. 2. On the left is the not-CQRS version of the service, and on the right is the CQRS version. [9]

Command Query Responsibility Segregation, as the name suggests, is all about segregation, or the separation of concerns. As figure 2 shows, in a non-CQRS service, the create, read, update and delete (CRUD) operations are typically implemented by a domain model that's mapped to a database. An Object-Relational Mapping (ORM) [8] is a common technique in these cases, although for performance reasons, a few queries might bypass the domain model and access the database directly. On the other hand, a CQRS service splits a persistent data model and the modules that use it into two parts: the command side and the query side.

- The write side (or command side) implements create, update, and delete operations (CUD) managing its own database and ensuring business rules. It may also handle simple queries, such as non-join, primary key-based

[1]https://www.getambassador.io/

queries. The command side publishes events whenever its data changes;

- The read side (or query side) is separate to the write side, which means different object models, probably running in different processes, perhaps on separate hardware. The query side handles only read operation. It subscribes to events produced by the write side and uses them to build and maintain a model that is suitable for answering client queries. Consequently, it uses whatever kind of database makes sense for the queries that it must support. The read side can be seen as an implementation of the Reporting Database pattern [10].

CQRS enables the efficient implementation of queries that retrieve data owned by multiple services, especially compared to the API composition pattern which sometimes results in expensive, inefficient in-memory joins.

Furthermore, separation of the read and write sides allows each to be scaled appropriately to match the workload. For example, query sides typically encounter a much higher demand than the write one.

CQRS also allows an application or service to efficiently implement a diverse set of queries. Attempting to support all queries using a single persistent data model is also often challenging and in some cases impossible. Even when a relational database has extensions to support a particular kind of query, using a specialized database is often more efficient. The CQRS pattern avoids the limitations of a single database by defining one or more read sides, each of which efficiently implements specific queries.

Another benefit is that CQRS defines separate code modules and database schemas for the command and query sides of a service. By separating concerns, the command side and query side are independent from each other and likely to be simpler and easier to maintain. The write side database can be refactored without needing to change to read databases.

Even though CQRS has several benefits, there are the significant additional operational complexity of managing and operating the extra data stores, which might even be of different types. Besides, as you might expect, there's a delay between when the command side publishes an event and when the query side processes that event and updates its read model. A client application that updates a model using a write operation and then immediately queries a view may see the previous version of the data. It is likely though that the system has settled for eventual consistency, discussed in next section about distributed transactions.

On the implementation side, applying the CQRS pattern does not actually mandate that you use different data stores for write and read sides, or that you use any particular persistence technology such as a relational database, NoSQL store, or event store. Nevertheless, a NoSQL database is often a good choice for a CQRS read side, which can leverage its strengths like ease of scaling and ignore its weaknesses. A CQRS view is unaffected by the limitations of a NoSQL database, because it only uses simple transactions and executes a fixed set of queries without the need for relational joins. Actually you don't need to normalize the database, because it's read-only and the records can duplicate data as much as needed to make queries easier.

Although CQRS does not require messaging, it's common to use messaging to process commands and publish update events. Messages also allow CQRS write part to handle commands asynchronously from a queue, rather than being processed synchronously, making the service itself asynchronous.

In that case, the application must handle message failures or duplicate messages. A command or event handler should be idempotent whenever possible, that is duplicate events results in the same correct outcome. For instance, an events which cancels an order can be repeated multiple times with the same result. In other cases events can be structured differently to be idempotent.

```
{
  "likingUser": 2,
  "restaurantId": 1
}
```

A service could be subscribed to this `ThumbsUpEvent` event and increment a thumbs-up counter for a restaurant, but if the same event will arrive twice the the counter will be incremented twice as well. To solve this problem, the event structure can be changed as follows:

```
{
  "likingUser": 2,
  "restaurantId": 1,
  "totalLikes": 320
}
```

Thanks to that, no matter how many times event will arrive, the resulting state will be the same as only one copy arrived.

Unfortunately idempotent events make your system vulnerable for simultaneous events. Two `ThumbsUpEvent` could be published at the same time by two different users, however they will carry the same `totalLikes`. Out-of-order events are likewise troublesome: two events could be published sequentially but if they will arrive in the wrong order then the system will remain inconsistent.

As demonstrated, an idempotent event is not feasible most of the time. A non-idempotent event handler must instead detect and discard duplicate events by recording the ids of events that it has already processed. One viable solution is to keep the history of IDs of all events that they have already seen. Theoretically, a CQRS read side should keep a full history of events, but in practice, the last couple of hours should be sufficient.

CQRS provide a lot of benefits when it comes to the scalability of data queries. Unfortunately, it doesn't come for free and the attached costs may be severely underestimated causing significant decrease in productivity .

However, since we have seen how to query data spanning multiple services, we proceed to analyse more complex transactions that involve several services.

III. Distributed transactions

ACID (Atomicity, Consistency, Isolation, Durability) transactions greatly simplify the job of the developer by providing the illusion that each transaction has exclusive access to the data. In a microservice architecture, a single service can still use ACID transactions for local operations. The challenge, however, lies in implementing transactions for operations that update data owned by multiple services, each of which has its own database, without violating consistency.

By consistency, we mean state consistency that is a transaction preserves all the database rules, such as referential integrity or, for example, when transferring money from one account to another the total amount held in both accounts should not change. This is not to be confused with the copy consistency in the CAP theorem [15], which refers to having all the copies of the data up-to-date, a subset of the former consistency.

The traditional approach to maintaining data consistency across multiple services or databases is to use distributed transactions. The de facto standard for distributed transaction management is the X/Open Distributed Transaction Processing (DTP) Model [16], which uses two-phase commit (2PC) to ensure that all participants in a transaction either commit or rollback.

Several problems arise however when a 2PC protocol is used in a system where network and hardware failures occur or a server could just crash. For instance, the coordinator, as well as the participants, have states in which they block waiting for incoming messages [14]. For that reason, the main problem with traditional distributed transactions is that they reduce availability. For a distributed transaction to commit, all the participating services must be available.

The overall availability is the product of the availability of all of the participants in the transaction. If a distributed transaction involves four services that are 99.5% available, then the overall availability is 98%, which is significantly less. Each additional service involved in a distributed transaction further reduces availability.

On top of that, the famous CAP theorem provides another proof of the unsuitability of distributed transactions:

> It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: consistency, availability, partition tolerance.
>
> Eric Brewer

The original theorem refers to copy consistency, but ACID consistency also cannot be maintained across partitions.

Although the formulation was misleading and has been revisited [15], it remains true that the CAP theorem prohibits perfect availability and consistency in the presence of network partitions. But the three properties are continuous values other than binary. Availability is already expressed as a range from 0 to 100%, but there are also many levels of consistency and even partitions usually affect portions of a system, not a globally.

The modern CAP goal tries to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition to mantain availability and for recovery afterward, for keep consistency.

As a matter of fact, many systems nowadays drop perfect consistency in favour of eventual consistency [17], availability and partition tolerance. Temporary inconsistency can be tolerated for two reasons: for improving read and write performance under highly concurrent conditions and for handling partition cases where a majority model would render part of the system unavailable even though the nodes are up and running. Clearly, partition recovery will need to restore ACID consistency since maintaining invariants during partitions might be impossible.

It is clear at this point that to solve the complex problem of maintaining data consistency in a microservice architecture, an application must use a different mechanism that builds on top of the asynchronous CQRS services. CQRS services are already partition tolerant since commands and queries are handled without the need of external services and events are usually implemented as asynchronous messages. What is missing though is a mechanism to keep the consistency and plan for recovery if a partition occurs. This is where the saga pattern comes in.

IV. The saga pattern

A saga is a sequence of local transactions. Each local transaction updates data within a single service using the familiar ACID transaction.

A saga is initiated by a command event and it reacts to subsequent command events, generating new ones and thus allowing command handlers to be kept independent.

While handling a request, the service doesn't synchronously interact with any other service. Instead, it asynchronously sends messages to other services which are not required to be available at the same time. Eventually, any unavailable service will come back up and process queued messages.

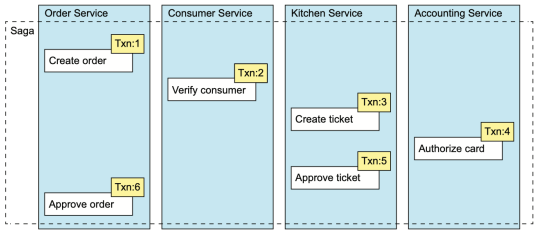

Fig. 3. Creating an Order using a saga. [11]

The example saga used throughout this section is the `Create Order Saga`, which is shown in Figure 3. The saga's first local transaction is initiated by a command event to create an order. The other five local transactions are each triggered by command events signalling the completion of the previous one.

This saga consists of the following local transactions:

1) Order Service: it creates an order in an `APPROVAL_PENDING` state;
2) Consumer Service: it verifies that the consumer can place an order;
3) Kitchen Service: it validates order details and create a ticket in the `CREATE_PENDING` state;
4) Accounting Service: it authorises the consumer's credit card;
5) Kitchen Service: it changes the state of the ticket to `AWAITING_ACCEPTANCE`;
6) Order Service: it changes the state of the order to `APPROVED`.

A. Compensating transactions

A great feature of traditional ACID transactions is that the business logic can easily roll back a transaction if it detects the violation of a business rule. Unfortunately, sagas can't be automatically rolled back, because each step commits its changes to the local database.

To see how compensating transactions are used, imagine a scenario where the authorisation of the consumer's credit card fails. In this scenario, the saga executes the following local transactions:

1) Order Service: it creates an order in an `APPROVAL_PENDING` state;
2) Consumer Service: it verifies that the consumer can place an order;
3) Kitchen Service: it validates order details and creates a ticket in the `CREATE_PENDING` state;
4) Accounting Service: it authorises consumer's credit card, which fails;
5) Kitchen Service: it changes the state of the ticket to `CREATE_REJECTED`;
6) Order Service: it changes the state of the order to `REJECTED`.

The fifth and sixth steps are compensating transactions that undo the updates made by the Kitchen Service and Order Service, respectively.

A saga's coordination logic is therefore responsible for sequencing the execution of both forward and compensating transactions. [11]

B. Lack of isolation

The I in ACID stands for isolation. The isolation property of ACID transactions ensures that the outcome of executing multiple transactions concurrently is the same as if they were executed in some serial order. The database provides the illusion that each ACID transaction has exclusive access to the data.

The challenge of using sagas is that they lack the isolation property of ACID transactions. That's because the updates made by each of a saga's local transactions are immediately visible to other sagas once that transaction commits. This behaviour can cause two problems. First, other sagas can change the data accessed by the saga while it's executing. And second, other sagas can read its data before the saga has completed its updates and consequently can be exposed to inconsistent data.

This lack of isolation potentially causes database *anomalies*. An anomaly is when a transaction reads or writes data in a way that it wouldn't if transactions were executed one at time. When an anomaly occurs, the outcome of executing sagas concurrently is different than if they were executed serially.

The lack of isolation can cause the following three anomalies [12]:

1) *Lost updates*: one saga overwrites without reading changes made by another saga;
2) *Dirty reads*: a transaction or a saga reads the updates made by a saga that has not yet completed;
3) *Fuzzy/non-repeatable reads*: two different steps of one saga read the same data and get different results because another saga has made updates;

It's then the responsibility of the developer to adopt a set of countermeasures for handling anomalies caused by lack of isolation that either prevent one or more anomalies or minimize their impact on the business. [18]

One example of countermeasure is the usage of semantic locks: a saga's compensatable transaction sets a flag in any record that it creates or updates. The flag indicates that the record isn't committed and could potentially change. The flag can either be a lock that prevents other transactions from accessing the record or a warning that indicates that other transactions should treat that record with suspicion.

C. Reliable events

It's essential for the database update and the publishing of the event to happen atomically. Consequently, to communicate reliably, the saga participants must use transactional messaging. Otherwise, a service might update the database and then crash, for example, before sending the message. If the service doesn't perform these two operations atomically, a failure could leave the system in an inconsistent state.

A straightforward way to reliably publish messages is to apply the Transactional outbox pattern. This pattern uses a database table as a temporary message queue, where the service inserts messages. Atomicity is guaranteed because this is a local ACID transaction. Then, an effortless way to publish the messages is to poll the table for unpublished messages although frequently polling the database can be expensive.

A sophisticated solution is to tail the database transaction log (also called the commit log). Every committed update made by an application is represented as an entry in the database's transaction log. A transaction subscriber can read the transaction log and publish each change as a message to the message broker. [19]

V. Communication

As we have seen with the previous patterns, the services within a microservice architecture must frequently collaborate in order to complete a saga or to handle a query request. Because service instances are typically processes running

on multiple machines, they must interact using Inter-Process Communication (IPC).

The choice of IPC mechanism is an important architectural decision since it can impact application availability and it intersects with transaction management. As we have seen with the saga pattern, we favour loosely coupled services that communicate with one another using asynchronous messaging.

Synchronous protocols such as REST and RPC are used mostly to communicate with other applications. The problem with REST is that it's a synchronous protocol: an HTTP client must wait for the service to send a response. Whenever services communicate using a synchronous protocol, the availability of the application is reduced.

A. Message Queuing

On the other hand, the basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues by using a message broker, an infrastructure service through which all messages flow.

A critical aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue. No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behaviour of the recipient. These semantics permit communication to be *loosely coupled in time*.

When selecting a message broker, you have various factors to consider, including the following:

- Messaging ordering: does the message broker preserve ordering of messages?
- Delivery guarantees: what kind of delivery guarantees does the broker make?
- Persistence: are messages persisted to disk and able to survive broker crashes?
- Durability: if a consumer reconnects to the message broker, will it receive the messages that were sent while it was disconnected?
- Scalability: how scalable is the message broker?
- Latency: what is the end-to-end latency?

Each broker makes different trade-offs. For example, a very low-latency broker might not preserve ordering, make no guarantees to deliver messages and only store messages in memory. A messaging broker that guarantees delivery and reliably stores messages on disk will probably have higher latency. Which kind of message broker is the best fit depends on your application's requirements. [13]

After picking the proper broker solution for your application, we can see how the saga orchestrator communicates with the participants using async reply-style interaction based on message queues. To execute a saga step, it sends a command message to a participant telling what operation to perform. After the saga participant has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which saga step to perform next.

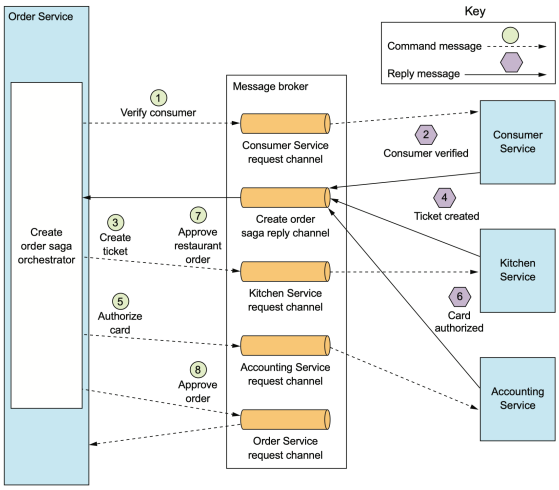

Fig. 4. Order Service implements a saga orchestrator, which invokes the saga participants using asynchronous request/response. [11]

Although CQRS does not require messaging, it's common to use messaging to process commands and publish update events. Clients typically send commands to the domain through a messaging system such as a queue. Messages also allow CQRS write part to handle commands asynchronously from a queue, rather than being processed synchronously.

VI. Technologies

https://github.com/chimurai/http-proxy-middleware but doesn't have rate limiting and other edge functions

A. Reactive Programming

Example of using forkJoin for OrderDetails API composition.

B. Promises and Futures

Error handling compared to callbacks.

Performance problems of Reactive streams and Promises.

C. Event bus

VII. Conclusion

We started the paper by showing the reasons behind a database-per-service solution in a microservice architecture. Unfortunately, no technology is a silver bullet and this solution has several significant drawbacks and it adds risky complexity.

One issue is that developers must deal with the additional complexity of managing databases in a distributed system. It's crucial to use an interprocess communication pattern aware of the asynchronous nature of the communication and the transience of the system, such as message queuing. Missing a message like a command event could indeed leave the system in an inconsistent state.

Moreover, implementing use cases that span multiple services requires the use of unfamiliar techniques like CQRS and sagas. One big obstacle that developers will have to face when adopting microservices is moving from a single database with

ACID transactions to a multi-database architecture with ACD sagas, because they're used to the simplicity of the ACID transaction model.

All the patterns explored in this paper require a significant mental leap, so they shouldn't be tackled unless the benefit is worth the jump. Having a database-per-service might be needed on specific portions of a system and not the system as a whole, based on the different scaling/usage characteristics. In this way, some portions can use a shared database without encountering all the complexities seen before, while others will benefit from a separated data store and the previous patterns. The former services may no be truly independent microservices, more just services in Service-Oriented-Architecture [20], but it is crucial for development teams to understand how to balance the cost and benefits of microservices.

REFERENCES

[1] Alberto Simioni, Tullio Vardanega, "In Pursuit of Architectural Agility: Experimenting with Microservices" in *2018 IEEE International Conference on Services Computing (SCC)*, July 2018. [Online] Available: https://ieeexplore.ieee.org/document/8456408

[2] Randy Shoup, "The eBay Architecture: Striking a Balance between Site Stability, Feature Velocity, Performance, and Cost", pp. 21-22. [Online]. Available: https://www.slideshare.net/RandyShoup/the-ebay-architecture-striking-a-balance-between-site-stability-feature-velocity-performance-and-cost

[3] Chris Richardson, "Database per service" in "Microservice Architecture". [Online]. Available: https://microservices.io/patterns/data/database-per-service.html

[4] Chris Richardson, "Querying using the API composition pattern" in "Microservices Patterns", pp. 221-228, October 2018.

[5] Microsoft Developer Network, "Gateway Aggregation pattern" in "Cloud Design Patterns". [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation

[6] Kisalaya Prasad, Avanti Patil, Heather Miller, "Futures and Promises" in "Programming Models for Distributed Computing". [Online]. Available: http://dist-prog-book.com/chapter/2/futures.html

[7] Martin Fowler, "CQRS". [Online]. Available: http://hibernate.org/orm/what-is-an-orm/

[8] Hibernate, "What is Object/Relational Mapping". [Online]. Available: https://martinfowler.com/bliki/CQRS.html

[9] Chris Richardson, "Overview of CQRS" in "Microservices Patterns", pp. 232-233, October 2018.

[10] Martin Fowler, "ReportingDatabase". [Online]. Available: https://martinfowler.com/bliki/ReportingDatabase.html

[11] Chris Richardson, "Using the Saga pattern to maintain data consistency" in "Microservices Patterns", pp. 114-117, October 2018.

[12] Chris Richardson, "Handling the lack of isolation" in "Microservices Patterns", pp. 126-131, October 2018.

[13] Chris Richardson, "Overview of broker-based messaging" in "Microservices Patterns", pp. 92, October 2018.

[14] Maarten van Steen, Andrew S. Tanenbaum, "Distributed Systems", pp. 483-490, February 2017.

[15] Eric Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed". [Online]. Available: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/

[16] Microsoft Developer Network, "X-Open Distributed Transaction Processing Standard". [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms686548(v%3Dvs.85)

[17] Werner Vogels, "Eventually Consistent". [Online]. Available: https://queue.acm.org/detail.cfm?id=1466448

[18] Lars Frank, Torben U. Zahle, " Semantic ACID properties in multi-databases using remote procedure calls and update propagations". [Online]. Available: https://dl.acm.org/citation.cfm?id=284472.284478

[19] Oded Shopen, "Listen to Yourself: A Design Pattern for Event-Driven Microservices". [Online]. Available: https://medium.com/@odedia/listen-to-yourself-design-pattern-for-event-driven-microservices-16f97e3ed066

[20] Microsoft Developer Network, "Service Oriented Architecture (SOA)" in "SOA in the Real World". [Online]. Available: https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx