Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 2

This item in chinese

Key Takeaways

- Event Sourcing is a technique for reliably updating state and publishing events that overcomes limitations of other solutions.
- The design concepts for an event-driven architecture, using event sourcing, align well with microservices architecture patterns.
- Snapshots can improve performance when querying aggregates by combining all events up to a certain point in time.
- Event sourcing can create challenges for queries, but these are overcome by following CQRS guidelines and materialized views.
- Event sourcing and CQRS do not require any specific tools or software, and many frameworks exist which can fill in some of the low-level functionality.

Part 1 described how a key obstacle in using the microservice architecture is that domain models, transactions and queries are surprisingly resistant to functional decomposition. It showed that the solution is to implement the business logic for each service as a set of DDD aggregates. Each transaction updates or creates a single aggregate. Events are used to maintain data consistency between aggregates (and services).

In Part 2, we describe how a key challenge with using events is atomically updating an aggregate and publishing an event. We show how to solve this problem by using event sourcing, which is an event-centric approach to business logic design and persistence. After that, we describe how the microservice architecture makes it difficult to implement queries. We show how an approach called Command Query Responsibility Segregation (CQRS) can implement scalable and performant queries.

Reliably Updating State and Publishing Events

On the surface, using events to maintain consistency between aggregates seems quite straightforward. When a service creates or updates an aggregate in the database it simply publishes an event. But there is a problem: updating the database and publishing an event must be done atomically. Otherwise, if, for example, a service crashed after updating the database but before publishing an event then the system would remain in an inconsistent state. The traditional solution is a distributed transaction involving the database and the message broker. But, for the reasons described earlier in part 1, 2PC is not a viable option.

There are a few different ways to solve this problem without using 2PC. One solution, which is shown in figure 1, is for the application to perform the update by publishing an event to a message broker such as Apache Kafka. A message consumer that subscribes to message broker eventually updates the database. This approach guarantees that the database is updated and the event is published. The drawback is that it implements a much more complex consistency model. An application cannot immediately read its own writes.

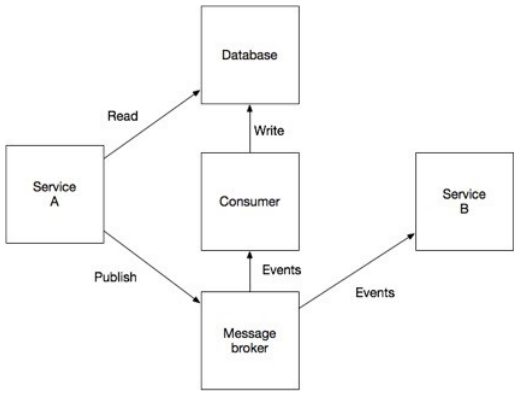

Figure 1 - Updating the database by publishing to a message broker

Another option, which is shown in figure 2, is for the application to tail the database transaction log (a.k.a. commit log), transform each recorded change into an event, and publish that event to the message broker. An important benefit of this approach is that it doesn't require any changes to the application. One drawback, however, is that it can be difficult to reverse engineer the high-level business event - the reason for the database update - from the low-level changes to the rows in the tables.

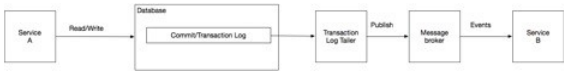

Figure 2 - Tailing the database transaction log

The third solution, which is shown in figure 3, is to use a database table as a temporary message queue. When a service updates an aggregate, it inserts an event into an EVENTS database table as part of the local ACID transaction. A separate process polls the EVENTS table and publishes events to the message broker. A nice feature of this solution is that the service is able to publish high-level business events. The downside is that it is potentially error-prone since the event publishing code must be synchronized with the business logic.

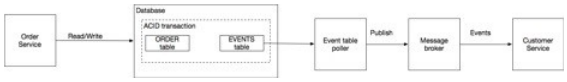

Figure 3 - Using a database table as a message queue

All three options have significant drawbacks. Publishing to a message broker and updating later doesn't provide read-your-writes consistency. Tailing the transaction log provides consistent reads but can't always publish high-level business events. Using a database table as a message queue provides consistent reads and publishes high-level business events but it relies on the developer remembering to publish an event when state changes. Fortunately, there is another option. It is an event-centric approach to persistence and business logic known as event sourcing.

Developing Microservices with Event Sourcing

Event sourcing is an event-centric approach to persistence. It is not a new idea. I first learned about event sourcing 5+ years ago, but it remained a curiosity until I started developing microservices. That is because, as you will see, event sourcing is a great way to implement an event-driven microservices architecture.

A service that uses event sourcing persists each aggregate as a sequence of events. When it creates or updates an aggregate, the service saves one or more events in the database, which is also known as the event store. It reconstructs the current state of an aggregate by loading the events and replaying them. In functional programming terms, a service reconstructs the state of an aggregate by performing a functional fold/reduce over the events. Because the events are the state, you no longer have the problem of atomically updating state and publishing events.

Consider, for example, the Order Service. Rather than store each Order as a row in an ORDERS table, it persists each Order aggregate as a sequence of events Order Created, Order Approved, Order Shipped, etc.. Figure 4 shows how these events might be stored in an SQL-based event store.

EVENTS

event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...	...	...	...	...

Figure 4 - Persisting an Order using event sourcing

The purpose of each column is as follows:

- entity_type and entity_id columns - identify the aggregate
- event_id - identify the event
- event_type - the type of the event
- event_data - the serialized JSON representation of the event's attributes

Some events contain a lot of data. The Order Created event, for example, contains the complete order including its line items, payment information and delivery information. Other events, such as the Order Shipped event, contain little or no data and just represent the state transition.

Event Sourcing and Publishing Events

Strictly speaking, event sourcing simply persists aggregates as events. It is straightforward, however, to also use it as a reliable event publishing mechanism. Saving an event is an inherently atomic operation that guarantees that the event store will deliver the event to services that are interested. If, for example, events are stored in the EVENTS table shown above, subscribers can simply poll the table for new events. More sophisticated event stores will use a different approach that has similar guarantees but is more performant and scalable. For example, Eventuate Local uses transaction log tailing. It reads events inserted into the EVENTS table from the MySQL replication stream and publishes them to Apache Kafka.

Using Snapshots to Improve Performance

An Order aggregate has relatively few state transitions and so it only has a small number of events. It is efficient to query the event store for those events and reconstruct an Order aggregate. Some aggregates, however, have a large number of events. For example, a Customer aggregate could potentially have a lots of Credit Reserved events. Over time, it would become increasingly inefficient to load and fold those events.

A common solution is to periodically persist a snapshot of the aggregate's state. The application restores the state of an aggregate by loading the most recent snapshot and only those events that have occurred since the snapshot was created. In functional terms, the snapshot is the initial value of the fold. If an aggregate has a simple, easily serializable structure then the snapshot can simply be, for example, its JSON serialization. More complex aggregates can be snapshotted by using the Memento pattern.

The Customer aggregate in the online store example has a very simple structure : the customer's information, their credit limit and their credit reservations. A snapshot of a Customer is simply the JSON serialization of its state. Figure 5 shows how to recreate a Customer from a snapshot corresponding to the state of a Customer as of event #103. The Customer Service just needs to load the snapshot and the events that have occurred after event #103.

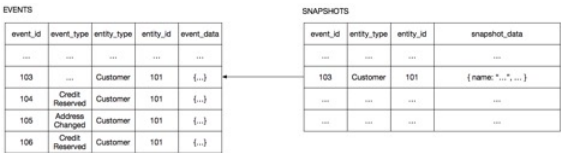

Figure 5 - Using snapshots to optimize performance

The Customer Service recreates the Customer by deserializing the snapshot's JSON and then loading and applying events #104 through #106.

Implementing Event Sourcing

An event store is a hybrid of a database and a message broker. It is a database because it has an API for inserting and retrieving an aggregate's events by primary key. An event store is also a message broker since it has an API for subscribing to events.

There are a few different ways to implement an event store. One option is to write your own event sourcing framework. You can, for example, persist events in an RDBMS. A simple, albeit low performance way to publish events is for subscribers to poll the EVENTS table for events.

Another option is to use a special purpose event store, which typically provides a rich set of features and better performance and scalability. Greg Young, an event sourcing pioneer, has a .NET-based, open-source event store called Event Store. Lightbend, the company formerly known as Typesafe, has a microservices framework called Lagom that is based on event sourcing. My startup, Eventuate, has a event sourcing framework for microservices that is available as a cloud service and a Kafka/RDBMS-based open-source project.

Benefits and Drawbacks of Event Sourcing

Event sourcing has both benefits and drawbacks. A major benefit of event sourcing is that it reliably publishes events whenever the state of an aggregate changes. It is a good foundation for an event-driven microservices architecture. Also, because each event can record the identity of the user that made the change, event sourcing provides an audit log that is guaranteed to be accurate. The stream of events can be used for a variety of other purposes including sending notifications to users, and application integration.

Another benefit of event sourcing is that it stores the entire history of each aggregate. You can easily implement temporal queries that retrieve the past state of an aggregate. To determine the state of an aggregate at a given point in time you simply the fold the events that occurred up until that point. It is straightforward to, for example, calculate the available credit of a customer at some point in the past.

Event sourcing also mostly avoids the O/R impedance mismatch problem. That is because it persists events rather than aggregates. Events typically have a simple, easily serializable, structure. A service can snapshot a complex aggregate by serializing a memento of its state. The Memento pattern adds a level of indirection between an aggregate and its serialized representation.

Event sourcing is, of course, not a silver bullet and it has some drawbacks. It is a different and unfamiliar programming model so there is a learning curve. In order for an existing application to use event sourcing, you must rewrite its business logic. Fortunately, this is a fairly mechanical transformation, which can be done when you migrate your application to microservices.

Another drawback of event sourcing it that message brokers usually guarantee at-least once delivery. Event handlers that are not idempotent must detect and discard duplicate events. The event sourcing framework can help by assigning each event a monotonically increasing id. An event handler can then detect duplicate events by tracking of highest seen event ids.

Another challenge with event sourcing is that the schema of events (and snapshots!) will evolve over time. Since events are stored forever, a service might need to fold events corresponding to multiple schema versions when it reconstructs an aggregate. One way to simplify a service is for the event sourcing framework to transform all events to the latest version of the schema when it loads them from the event store. As a result, a service only needs to fold the latest version of the events.

Another drawback of event sourcing is that querying the event store can be challenging. Let's imagine, for example, that you need to find credit worthy customers who have a low credit limit. You cannot simply write SELECT * FROM CUSTOMER WHERE CREDIT_LIMIT < ? AND c.CREATION_DATE > ?. There isn't a column containing the credit limit. Instead, you must use a more complex and potentially inefficient query that has a nested SELECT to compute the credit limit by folding events that set the initial credit and adjust it. To make matters worse, a NoSQL-based event store will typically only support primary key-based lookup. Consequently, you must implement queries using an approach called Command Query Responsibility Segregation (CQRS).

Implementing Queries Using CQRS

Event sourcing is a major obstacle to implementing efficient queries in a microservice architecture. It isn't the only problem, however. Consider, for example, a SQL query that finds new customers that have placed high value orders.

```
SELECT *
FROM CUSTOMER c, ORDER o
WHERE
  c.id = o.ID
    AND o.ORDER_TOTAL > 100000
    AND o.STATE = 'SHIPPED'
    AND c.CREATION_DATE > ?
```

In a microservices architecture you cannot join the CUSTOMER and ORDER tables. Each table is owned by a different service and is only accessible via that service's API. You can't write traditional queries that join tables owned by multiple services. Event sourcing makes matters worse preventing you from writing simple, straightforward queries. Let's look at a way to implement queries in a microservice architecture.

Using CQRS

A good way to implement queries is to use an architectural pattern known as Command Query Responsibility Segregation (CQRS). CQRS, as the name suggests, splits the application into two parts. The first part is the command-side, which handles commands (e.g. HTTP POSTs, PUTs, and DELETEs) to create, update and delete aggregates. These aggregates are, of course, implemented using event sourcing. The second part of the application is the query side, which handles queries (e.g. HTTP GETs) by querying one or more materialized views of the aggregates. The query side keeps the views synchronized with the aggregates by subscribing to events published by the command side.

Each query-side view is implemented using whatever kind of database makes sense for the queries that it must support. Depending on the requirements, an application's query side might use one or more of the following databases:

Table 1. Query-side view stores

If you need....	then use....	for example...
PK-based lookup of JSON objects	**a document store such as MongoDB, or a key value store such as Redis.**	**Implement order history by maintaining a MongoDB Document for each customer that contains their orders**
Query-based lookup of JSON objects	**a document store such as MongoDB**	**Implement customer view using MongoDB**

Text queries	**a text search engine such as Elasticsearch**	**Implement text search for orders by maintaining a per-order Elasticsearch document**
Graph queries	**a graph database such as Neo4j**	**Implement fraud detection by maintaining a graph of customers, orders, and other data**
Traditional SQL reporting/BI	**an RDBMS**	**Standard business reports and analytics**

In many ways, CQRS is a event-based, generalization of the widely used approach of using RDBMS as the system of record and a text search engine, such as Elasticsearch, to handle text queries. CQRS uses a broader range of database types - not just a text search engine. Also, it updates a query-side view in near real-time by subscribing to events.

Figure 6 shows the CQRS pattern applied to the online store example. The Customer Service and the Order Service are command-side services. They provide APIs for creating and updating Customers and orders. The Customer View Service is a query-side service. It provides an API for querying customers.

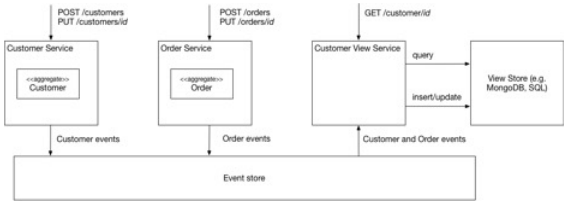

Figure 6 - Using CQRS in the online store

The Customer View Service subscribes to the Customer and Order events published by the command-side services. It updates a view store that is implemented using MongoDB. The service maintains a MongoDB collection of documents, one per customer. Each document has attributes for the customer details. It also has an attribute that stores the customer's recent orders. This collection supports a variety of queries including those described above.

Benefits and Drawback of CQRS

CQRS has both benefits and drawbacks. A major benefit of CQRS is that it makes it possible to implement queries in a microservices architecture, especially one that uses event sourcing. It enables an application to efficiently support a diverse set of queries. Another benefit is that the separation of concerns often simplifies the command and query sides of the application.

CQRS also has some drawbacks. One drawback is that it requires extra effort to develop and operate the system. You must develop and deploy the query-side services that update and query the views. You also need to deploy the view stores.

Another drawback of CQRS is dealing with the "lag" between the command side and the query side views. As you would expect, there is delay between when the query side is updated to reflect an update to a command-side aggregate. A client application that updates an aggregate and then immediately queries a view might see the previous version of the aggregate. It must often be written in a way that avoids exposing these potential inconsistencies to the user.

Summary

A major challenge when using events to maintain data consistency between services is atomically updating the database and publishing events. The traditional solution is to use a distributed transaction spanning the database and the message broker. 2PC, however, is not a viable technology for modern applications. A better approach is to use event sourcing, which is an event-centric approach to business logic design and persistence.

Another challenge in the microservice architecture is implementing queries. Queries often need to join data that is owned by multiple services. However, joins are no longer straightforward since data is private to each service. Using event sourcing also makes it even more difficult to efficiently implement queries since the current state is not explicitly stored. The solution is to use Command Query Responsibility Segregation (CQRS) and maintain one or more materialized views of the aggregates that can be easily queried.

About the Author

Chris Richardson is a developer and architect. He is a Java Champion and the author of POJOs in Action, which describes how to build enterprise Java applications with frameworks such as Spring and Hibernate. Chris was also the founder of the original CloudFoundry.com. He consults with organizations to improve how they develop and deploy applications and is working on his third startup. You can find Chris on Twitter @crichardson and on Eventuate.

Related Editorial

- **Microservices Framework Lagom 1.5 with Akka Management and Support for Kubernetes and OpenShift**
- **Migrating a Retail Monolith to Microservices: Sebastian Gauder at MicroXchg Berlin**
- **O'Reilly Publishes "The State of Microservices Maturity" Report**
- **Entity Services Increase Complexity: Tareq Abedrabbo Discusses Microservices Antipatterns**
- **QCon NY: Jonas Bonér on Designing Events-First Microservices**