Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 1

This item in chinese

Key takeaways

- The Microservice architecture functionally decomposes an application into services, each of which corresponds to a business capability.
- A key challenge when developing microservice-based business applications is that transactions, domain models, and queries resist decomposition.
- A domain model can be decomposed into Domain Driven Design aggregates.
- Each service's business logic is a domain model consisting of one or more Domain Driven Design aggregates.
- Within a service, each transaction creates or updates a single aggregate.
- Events are used to maintain consistency between aggregates (and services).

This article is a 2 part article. You can find Part 2 here.

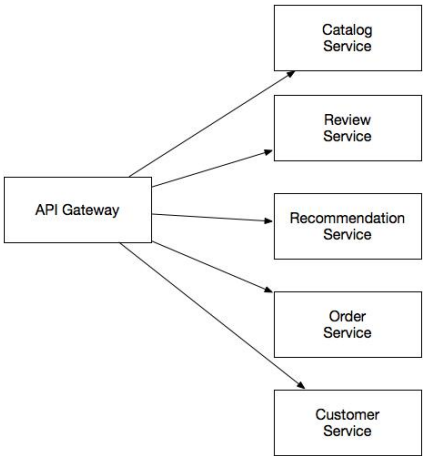
The microservice architecture is becoming increasingly popular. It is an approach to modularity that functionally decomposes an application into a set of services. It enables teams developing large, complex applications to deliver better software faster. They can adopt new technology more easily since they can implement each service with the latest and most appropriate technology stack. The microservices architecture also improves an application's scalability by enabling each service to be deployed on the optimal hardware.

Microservices are not, however, a silver bullet. In particular, domain models, transactions and queries are surprisingly resistant to functional decomposition. As a result, developing transactional business applications using the microservice architecture is challenging. In this article, I describe a way to develop microservices that solves these problems by using Domain Driven Design, Event Sourcing and Command Query Responsibility Segregation (CQRS). Let's first look at the challenges developers face when writing microservices.

Microservice Development Challenges

Modularity is essential when developing large, complex applications. Most modern applications are too large to be developed by an individual. They are also too complex to be understood by a single person. Applications must be decomposed into modules that are developed and understood by a team of developers. In a monolithic application, modules are defined using programming language constructs such as Java packages. However, this approach tends to not work well in practice. Long lived, monolithic applications usually degenerate into big balls of mud.

The microservice architecture uses services as the unit of modularity. Each service corresponds to a business capability, which is something an organization does in order to create value. A microservices-based online store, for example, consists of various services including Order Service, Customer Service, Catalog Service.

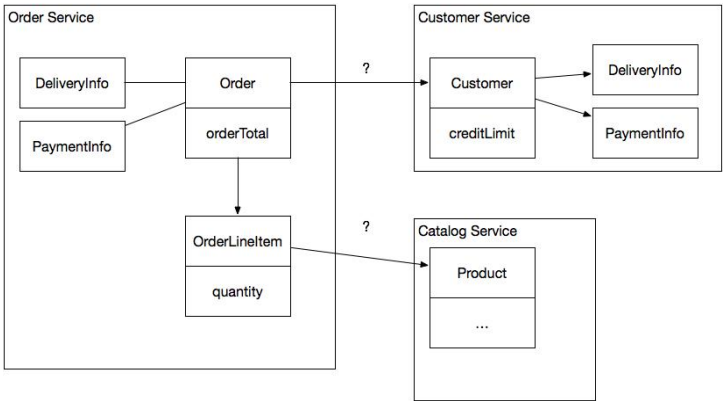


Each service has an impermeable boundary that is difficult to violate. As a result, the modularity of the application is much easier to preserve over time. The microservice architecture has other benefits including the ability to deploy and scale services independently.

Unfortunately, decomposing an application into services is not as easy as it sounds. Several different aspects of applications - domain models, transactions and queries - are difficult to decompose. Let's look at the reasons why.

Problem #1 - Decomposing a Domain Model

The Domain Model pattern is a good way to implement complex business logic. The domain model for an online store would include classes such as Order, OrderLineItem, Customer and Product. In a microservices architecture, the Order and OrderLineItem classes are part of the Order Service, the Customer class is part of the Customer Service, and the Product class belongs to the Catalog Service.



The challenge with decomposing the domain model, however, is that classes often reference one another. For example, an Order references its Customer and an OrderLineItem references a Product. What do we do about references that want to span service boundaries? Later on you will see how the concept of an Aggregate from Domain-Driven Design (DDD) solves this problem.

Microservices and Databases

A distinctive feature of the microservice architecture is that the data owned by a service is only accessible via that service's API. In the online store, for example, the OrderService has a database that includes the ORDERS table and the CustomerService has its database, which includes the CUSTOMERS table. Because of this encapsulation, the services are loosely coupled. At development time, a developer can change their service's schema without having to coordinate with developers working on other service. At runtime, the services are isolated from each other. For example, a service will never be blocked waiting for a database lock owned by another service. Unfortunately, the functional decomposition of the database makes it difficult to maintain data consistency and to implement many kinds of queries.

Problem #2 - Implementing Transactions That Span Services

A traditional monolithic application can rely on ACID transactions to enforce business rules (a.k.a. invariants). Imagine, for example, that customers of the online store have a credit limit that must be checked before creating a new order. The application must ensure that potentially multiple concurrent attempts to place an order do not exceed a customer's credit limit. If Orders and Customers reside in the same database it is trivial to use an ACID transaction (with the appropriate isolation level) as follows:

```
BEGIN TRANSACTION
…
SELECT ORDER_TOTAL
 FROM ORDERS WHERE CUSTOMER_ID = ?
…
SELECT CREDIT_LIMIT
FROM CUSTOMERS WHERE CUSTOMER_ID = ?
…
INSERT INTO ORDERS …
…
COMMIT TRANSACTION
```

Sadly, we cannot use such a straightforward approach to maintain data consistency in a microservices-based application. The ORDERS and CUSTOMERS tables are owned by different services and can only be accessed via APIs. They might also be in different databases.

The traditional solution is 2PC (a.k.a. distributed transactions) but this is not a viable technology for modern applications. The CAP theorem requires you to chose between availability and consistency, and availability is usually the better choice. Moreover, many modern technologies, such as most NoSQL databases, do not even support ACID transactions let alone, 2PC. Maintaining data consistency is essential so we need another solution. Later on you will see that the solution is to use an event-driven architecture based on a technique known as event sourcing.

Problem #3 - Querying and Reporting

Maintaining data consistency is not the only challenge. Another problem is querying and reporting. In a traditional monolithic application it is extremely common to write queries that use joins. For example, it is easy to find recent customers and their large orders using a query such as:

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

We cannot use this kind of query in a microservices-based online store. As mentioned earlier, the ORDERS and CUSTOMERS tables are owned by different services and can only be accessed via APIs. Some services might not even be using a SQL database. Others, as you will see below, might use an approach known as Event Sourcing, which makes querying even more challenging. Later on, you will learn that the solution is to maintain materialized views using an approach known as Command Query Responsibility Segregation (CQRS). But first, let's look at how Domain-Driven design (DDD) is an essential tool for the development of domain model-based business logic for microservices.

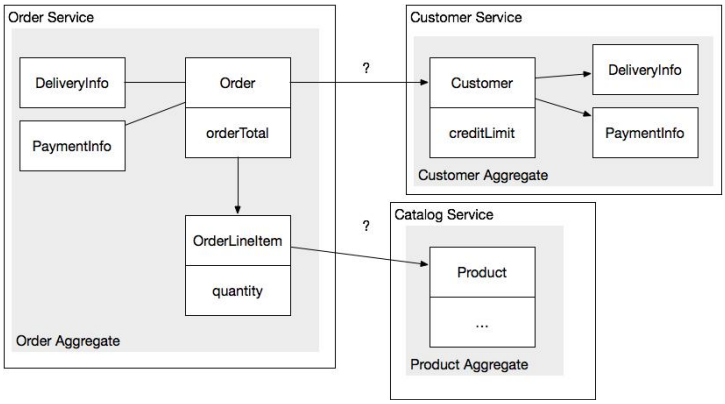
DDD Aggregates are the Building Blocks of Microservices

As you can see, there are several problems that must be solved in order to successfully develop business applications using the microservice architecture. The solution to some of these problems can be found in the must-read book Domain-Driven Design by Eric Evans. This book, published in 2003, describes an approach to designing complex software that is very useful when developing microservices. In particular, Domain-Driven Design enables you to create a modular domain model that can be partitioned across services.

What is an Aggregate?

In Domain-Driven Design, Evans defines several building blocks for domain models. Many have become part of everyday developer language including entity, which is an object with a persistent identity; value object, which is an object that has no identity and is defined by its attributes; service, which contains business logic that doesn't belong in an entity or value object service; and repository, which represents a collection of persistent entities. One building block, the aggregate, has mostly been ignored by developers except by those who are DDD purists. It turns out, however, that aggregates are key to developing microservices.

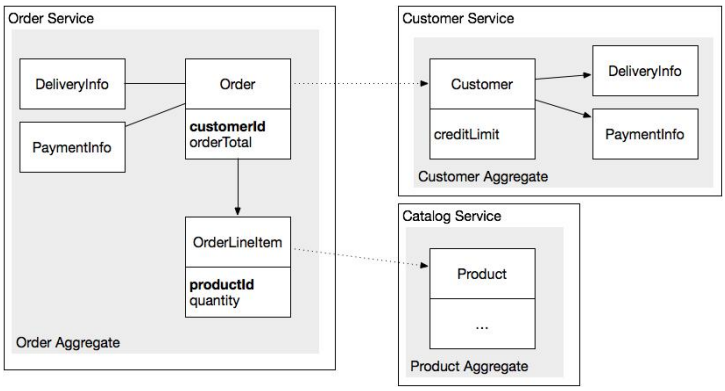
An aggregate is a cluster of domain objects that can be treated as a unit. It consists of a root entity and possibly one or more other associated entities and value objects. For example, the domain model for the online store contains aggregates such as Order and Customer. An Order aggregate consists of an Order entity (the root), one or more OrderLineItem value objects along with other value objects such as a delivery Address and PaymentInformation. A Customer aggregate consists of the Customer root entity along with other value objects such a DeliveryInfo and PaymentInformation.



Using aggregates decomposes a domain model into chunks, which are individually easier to understand. It also clarifies the scope of operations such as load and delete. An aggregate is usually loaded in its entirety from the database. Deleting an aggregate deletes all of the objects. The benefit of aggregates, however, goes far beyond modularizing a domain model. That is because aggregates must obey certain rules.

Inter-Aggregate References Must Use Primary Keys

The first rule is that aggregates reference each other by identity (e.g. primary key) instead of object references. For example, an Order references its Customer using a customerId rather than a reference to the Customer object. Similarly, an OrderLineItem references a Product using a productId.



This approach is quite different than traditional object modeling, which considers foreign keys in the domain model to be a design smell. The use of identity rather than object references means that the aggregates are loosely coupled. You can easily put different aggregates in different services. In fact, a service's business logic consists of a domain model that is a collection of aggregates. For example, the OrderService contains the Order aggregate and the CustomerService contains the Customer aggregate.

One Transaction Creates or Updates One Aggregate

The second rule that aggregates must obey is that a transaction can only create or update a single aggregate. When I first read about this rule many years ago, it made no sense! At the time, I was developing traditional monolithic, RDBMS-based applications and so transactions could update arbitrary data. Today, however, this constraint is perfect for the microservice architecture. It ensures that a transaction is contained within a service. This constraint also matches the limited transaction model of most NoSQL databases.

When developing a domain model, a key decision you must make is how large to make each aggregate. On the one hand, aggregates should ideally be small. It improves modularity by separating concerns. It is more efficient since aggregates are typically loaded in their entirety. Also, because updates to each aggregate happen sequentially, using fine grained aggregates will increase the number of simultaneous requests that the application can handle and so improve scalability. It will also improve the user experience since it reduces the likelihood of two users attempting to update the same aggregate. On the other hand, because an aggregate is the scope of a transaction, you might need to define a larger aggregate in order to make a particular update atomic.

For example, earlier I described how in the online store's domain model, Order and Customer are separate aggregates. An alternative design is to make Orders part of the Customer aggregate. A benefit of a larger Customer aggregate is that the application can enforce the credit check atomically. A drawback of this approach is that it combines order and customer management functionality into the same service. It also reduces scalability since transactions that update different orders for the same customer would be serialized. Similarly, two users might conflict if they attempted to edit different orders for the same customer. Also, as the number of orders grows it will become increasingly expensive to load a Customer aggregate. Because of these issues, it is best to make aggregates as fine-grained as possible.

Even though a transaction can only create or update a single aggregate, applications must still maintain consistency between aggregates. The Order Service must, for example, verify that a new Order aggregate will not exceed the Customer aggregate's credit limit. There are a couple of different ways to maintain consistency. One option is to cheat and create and/or update multiple aggregates in a single transaction. This is only possible if all aggregates are owned by the same service and persisted in same RDBMS. The other, more correct option is to maintain consistency between aggregates using an eventually consistent, event-driven approach.

Using Events to Maintain Data Consistency

In a modern application, there are various constraints on transactions that make it challenging to maintain data consistency across services. Each service has its own private data, yet 2PC is not a viable option. Moreover, many applications use NoSQL databases, which don't support local ACID transactions, let alone distributed transactions. Consequently, a modern application must use an event-driven, eventually consistent transaction model.

What is an Event?

According to Merriam-Webster an event is something that happens:

In this article, we define a domain event as something that has happened to an aggregate. An event usually represents a state change. Consider, for example, an Order aggregate in the online store. Its state changing events include Order Created, Order Cancelled, Order Shipped. Events can represent attempts to violate a business rule such as a Customer's credit limit.

Using an Event-Driven Architecture

Services use events to maintain consistency between aggregates as follows: an aggregate publishes an event whenever something notable happens, such as its state changing or there is an attempted violation of a business rule. Other aggregates subscribe to events and respond by updating their own state.

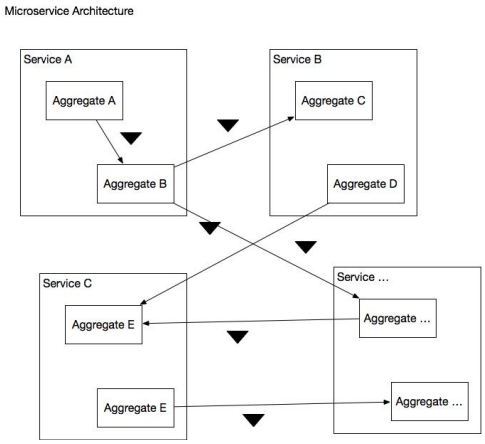
The online store verifies the customer's credit limit when creating an order using a sequence of steps:

1. An Order aggregate, which is created with a NEW status, publishes an OrderCreated event
2. The Customer aggregate consumes the OrderCreated event, reserves credit for the order and publishes an CreditReserved event
3. The Order aggregate consumes the CreditReserved event, and changes its status to APPROVED

If the credit check fails due to insufficient funds, the Customer aggregate publishes a CreditLimitExceeded event. This event does not correspond to a state change but instead represents a failed attempt to violate a business rule. The Order aggregate consumes this event and changes its state to CANCELLED.

Microservice Architecture as a Web of Event-Driven Aggregates

In this architecture, each service's business logic consists of one or more aggregates. Each transaction performed by a service updates or creates a single aggregate. The services maintain data consistency between aggregates by using events.



A distinctive benefit of this approach is that the aggregates are loosely coupled building blocks. They can be deployed as a monolith or as a set of services. At the start of a project you could use a monolithic architecture. Later, as the size of the application and the development team grows, you can then easily switch to a microservices architecture.

Summary

The Microservice architecture functionally decomposes an application into services, each of which corresponds to a business capability. A key challenge when developing microservice-based business applications is that transactions, domain models, and queries resist decomposition. You can decompose a domain model by applying the idea of a Domain Driven Design aggregate. Each service's business logic is a domain model consisting of one or more DDD aggregates.

Within each service, a transaction creates or updates a single aggregate. Because 2PC is not a viable technology for modern applications, events are used to maintain consistency between aggregates (and services). In part 2, we describe how to implement a reliable event-driven architecture using Event Sourcing. We also show how to implement queries in a microservice architecture using Command Query Responsibility Segregation.

This article is a 2 part article. You can find Part 2 here.

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

- **Migrating a Retail Monolith to Microservices: Sebastian Gauder at MicroXchg Berlin**
- **Are Frameworks Good or Bad, or Both?**
- **O'Reilly Publishes "The State of Microservices Maturity" Report**
- **The InfoQ eMag: Domain-Driven Design in Practice**
- **Readable Code - Why, How and When You Should Write It**