Business Logic in a Microservice Architecture

# Introduction

Developing complex business logic is even more challenging in a microservice architecture where the business logic is spread over multiple services.

You need to address two key challenges:

* First, a typical domain model is a tangled web of interconnected classes. Although this isn’t a problem in a monolithic application, in a microservice architecture, where classes are scattered around different services, **you need to eliminate object references that would otherwise span service boundaries.**
* The second challenge is designing business logic that works within the transaction management constraints of a microservice architecture. **Your business logic can use ACID transactions within services, but as described in chapter 4, it must use the Saga pattern to maintain data consistency across services.**

we can address these issues by using the Aggregate pattern from DDD. **The Aggregate pattern structures a service’s business logic as a collection of aggregates.**

**An *aggregate* is a cluster of objects inside a subdomain that can be treated as a single unit for data changes.**

There are two reasons why aggregates are useful when developing business logic in a microservice architecture:

 **Aggregates avoid any possibility of object references spanning service boundaries, because an inter-aggregate reference is a primary key value rather than an object reference.**

 **Because a transaction can only create or update a single aggregate, aggregates fit the constraints of the microservices transaction model.**

**As a result, an ACID transaction is guaranteed to be within a single service.**

* In a separate document we talked about organization of business logics in general and saw

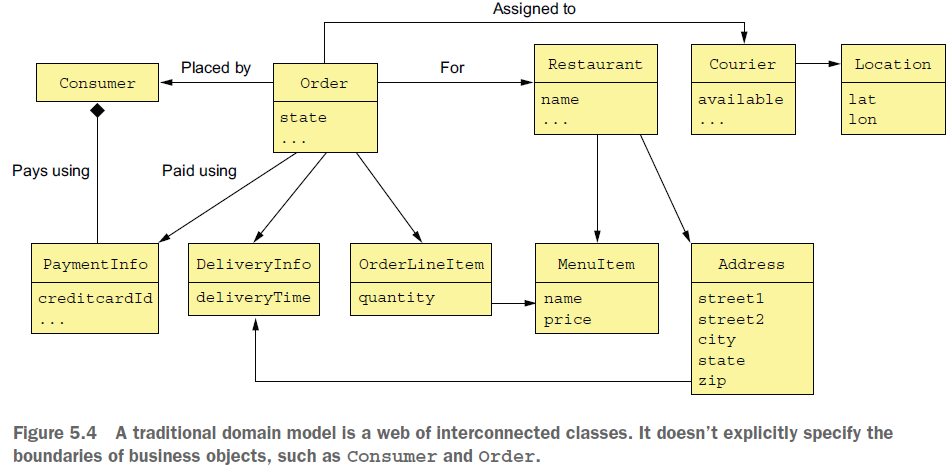
how an object-oriented approach called the domain model fits complex, evolving logics.

* In this document, we are going to talk about the problems with the classic OOD that the DDD approach as a refinement of the OOD will solve. We’ll introduce the concept of **a DDD aggregate** and explain why it’s a good building block for a service’s business logic.
* After that, I describe the Domain event pattern events and explain why it’s useful for a service to publish events. And finish up with some examples of implementing these concepts.

# Designing a Domain Model Using the DDD Aggregate Pattern

In traditional object-oriented design, a domain model is a collection of classes and relationships between classes. The classes are usually organized into packages.

figure 5.4 shows part of a domain model for the FTGO application. It’s a typical domain model consisting of a web of interconnected classes.



This example has several classes corresponding to business objects: Consumer, Order, Restaurant, and Courier. But interestingly, **the explicit boundaries of each business object are missing from this kind of traditional domain model.** It doesn’t specify, for example, which classes are part of the Order business object. This lack of boundaries can sometimes cause problems, especially in microservice architecture.

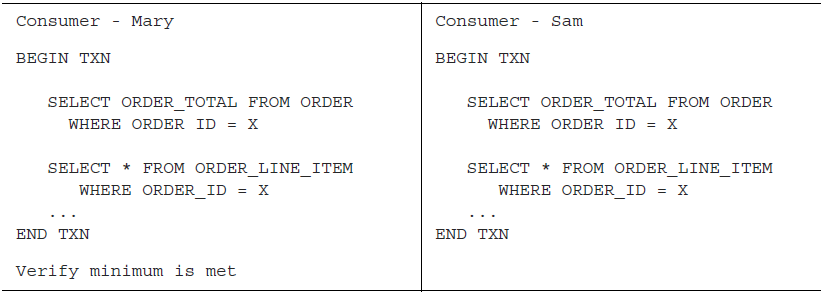
I begin this section with an example **problem caused by the lack of explicit boundaries**. Next, I describe the concept of **an aggregate and how it has explicit boundaries**. After that, I describe the **rules that aggregates must obey and how they make aggregates a good fit for the microservice architecture.** I then describe how to carefully choose the boundaries of your aggregates and why it matters. Finally, I discuss **how to design business logic using aggregates.**

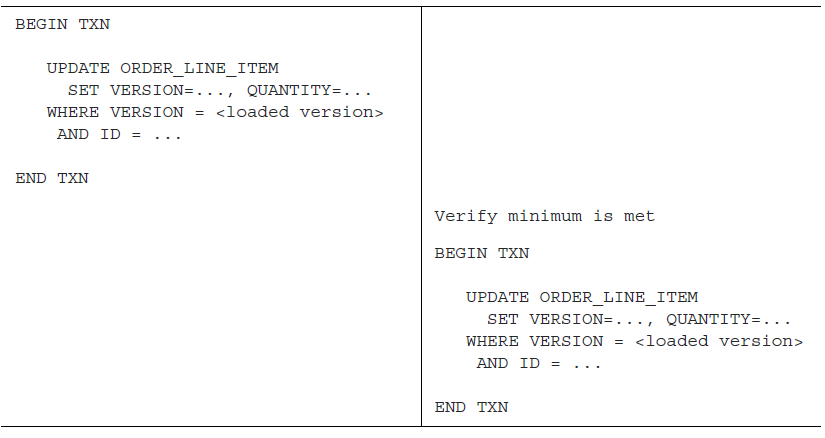
## The Problem with Fuzzy Boundaries

* Imagine, for example, that you want to perform an operation, such as a load or delete, on an Order business object. What exactly does that mean**? What is the scope an operation**? You would certainly load or delete the Order object. But in reality, there’s more to an Order than simply the Order object. There are also the order line items, the payment information, and so on. **Figure 5.4 leaves the boundaries of a domain object to the developer’s intuition.**
* Besides a conceptual fuzziness**, the lack of explicit boundaries causes problems when updating a business object.** A typical business object has ***invariants*, business rules that must be enforced at all times.** An Order has a minimum order amount, for example. The FTGO application must ensure that any attempt to update an order doesn’t violate an invariant such as the minimum order amount. The challenge is that **in order to enforce invariants, you must design your business logic carefully.**

For example, let’s look at how to ensure the order minimum is met when multiple consumers work together to create an order. Two consumers—Sam and Mary—are working together on an order and simultaneously decide that the order exceeds their budget. Sam reduces the quantity of samosas, and Mary reduces the quantity of naan

bread. From the application’s perspective, both consumers retrieve the order and its line items from the database. Both consumers then update a line item to reduce the cost of the order. From each consumer’s perspective the order minimum is preserved. Here’s the sequence of database transactions:





Each consumer changes a line item using a sequence of two transactions:

* The first transaction loads the order and its line items. The UI verifies that the order minimum is satisfied before executing the second transaction.
* The second transaction updates the line-item quantity using an optimistic offline locking check that verifies that the order line is unchanged since it was loaded by the first transaction.

In this scenario, Sam reduces the order total by $X and Mary reduces it by $Y. As a result, the Order is no longer valid, even though the application verified that the order still satisfied the order minimum after each consumer’s update. As you can see, **directly updating part of a business object can result in the violation of the business rules. DDD aggregates are intended to solve this problem.**

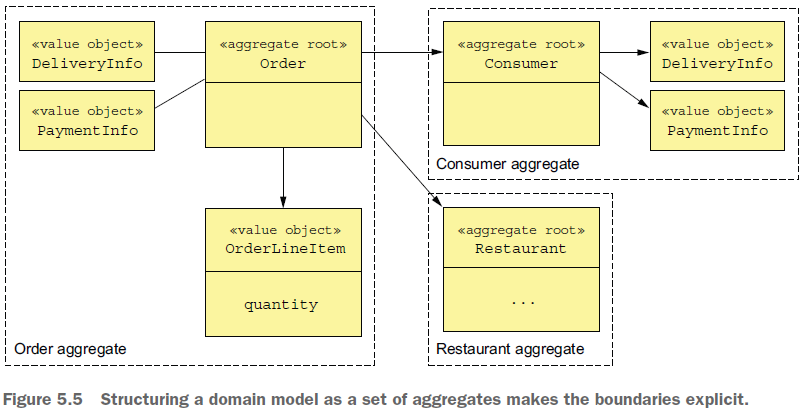
## Aggregates Have Explicit Boundaries

The Aggregate Pattern*: “Organize a domain model as a collection of aggregates, each of which is* ***a graph of objects that can be treated as a unit.”***

An *aggregate* is **a cluster of domain objects** **within a boundary** that **can be treated as a unit**. It consists of **a root entity** and possibly **one or more other entities** **and value objects**.

Many business objects are modeled as aggregates. For example, in chapter 2 we created a rough domain model **by analyzing the nouns used in the requirements and by domain experts**. Many of these nouns, such as Order, Consumer, and Restaurant, are aggregates.

Figure 5.5 shows the Order aggregate and its boundary. An Order aggregate consists of an Order entity, one or more OrderLineItem value objects, and other value objects such as a delivery Address and PaymentInformation:



**Aggregates decompose a domain model into chunks, which are individually easier to understand.** They also clarify the scope of operations such as load, update, and delete. **These operations act on the entire aggregate rather than on parts of it.**

**An aggregate is often loaded in its entirety from the database, thereby avoiding any complications of lazy loading. Deleting an aggregate, removes all of its objects from a database.**

### AGGREGATES ARE CONSISTENCY BOUNDARIES

Updating an entire aggregate rather than its parts solves the consistency issues, such as the example described earlier. **Update operations are invoked on the aggregate root, which enforces invariants. (Isn’t this supposed to imply the integrity of the data not the consistency?)**

Also, concurrency is handled by locking the aggregate root using, for example**, a version number** or a **database-level lock.**

For example, instead of updating line items’ quantities directly, a client must invoke a method on

the root of the Order aggregate, which enforces invariants such as the minimum order amount. Note, though, that this approach doesn’t require the entire aggregate to be updated in the database. An application might, for example, only update the rows corresponding to the Order object and the updated OrderLineItem.

### IDENTIFYING AGGREGATES IS KEY

In DDD, a key part of designing a domain model is **identifying aggregates**, **their boundaries**, and **their roots**. The details of the aggregates’ internal structure is secondary.

The benefit of aggregates, however, goes far beyond modularizing a domain model. That’s because aggregates must obey certain rules.

## Aggregate Rules

**These rules** **ensure that an aggregate is a self-contained unit that can enforce its invariants**. Let’s look at each of the rules:

### Rule #1: Reference Only the Aggregate Root

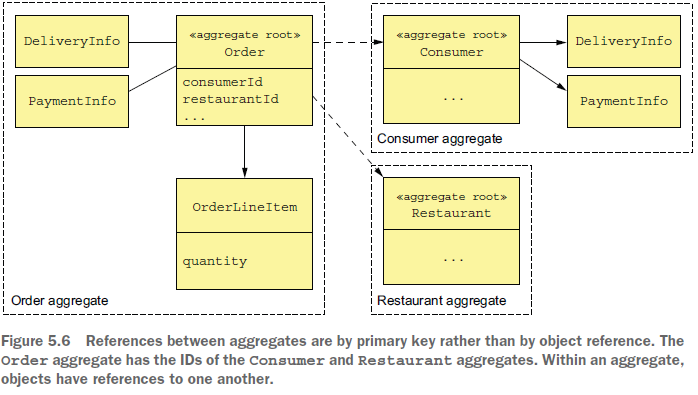
The previous example illustrated the perils of updating OrderLineItems directly. The goal of the first aggregate rule is to eliminate this problem. **It requires that the root entity be the only part of an aggregate that can be referenced by classes outside of the aggregate**.

“A client can only update an aggregate by invoking a method on the aggregate root.”

A service, for example, uses a repository to load an aggregate from the database and obtain a reference to the aggregate root. It updates an aggregate by invoking a method on the aggregate root. **This rule ensures that the aggregate can enforce its invariant.**

### Rule #2: Inter-Aggregate References Must Use Primary Keys

Another rule is that **aggregates reference each other by identity** (for example, primary key) **instead of object references**. For example, as figure 5.6 shows, an Order references its Consumer using a consumerId rather than a reference to the Consumer object. Similarly, an Order references a Restaurant using a restaurantId.



**This approach is quite different from traditional object modeling, which considers foreign keys in the domain model to be a design smell.** It has a number of benefits:

* **The use of identity** rather than object references **means that the aggregates are loosely coupled**.
  + It ensures that the aggregate boundaries between aggregates are well defined
  + and avoids accidentally updating a different aggregate.
  + Also, if an aggregate is part of another service, there isn’t a problem of object references that span services.
* **This approach also simplifies persistence since the aggregate is the unit of storage**. It makes it **easier to store aggregates in a NoSQL database such as MongoDB.**
* It also eliminates the need for transparent lazy loading and its associated problems.

**I add** : you need to explicitly fetch the associated aggregate using its Id instead of relying on proxies in a transparent way?

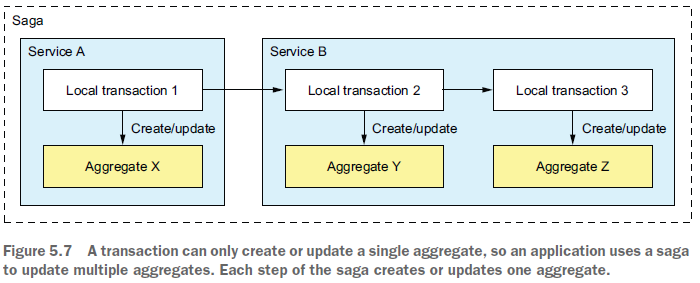
* **Scaling the database by sharding aggregates is relatively straightforward.**

### Rule #3: One Transaction Creates or Updates One Aggregate

Another rule that aggregates must obey is that a transaction can only create or update a single aggregate. When I first read about it many years ago, this rule made no sense! At the time, I was developing traditional monolithic applications that used an RDBMS, so transactions could update multiple aggregates. Today, 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.

This rule makes it more complicated to implement operations that need to create or update multiple aggregates. But this is exactly the problem that sagas (described in chapter 4) are designed to solve. **Each step of the saga creates or updates exactly one aggregate**. Figure 5.7 shows how this works.



In this example, the saga consists of three transactions. The first transaction updates aggregate X in service A. The other two transactions are both in service B. One transaction updates aggregate X, and the other updates aggregate Y.

An alternative approach to maintaining consistency across multiple aggregates within a single service is to cheat and update multiple aggregates within a transaction. For example, service B could update aggregates Y and Z in a single transaction. **This is only possible when using a database, such as an RDBMS, that supports a rich transaction model.** If you’re using a NoSQL database that only has simple transactions, there’s no other option except to use sagas.

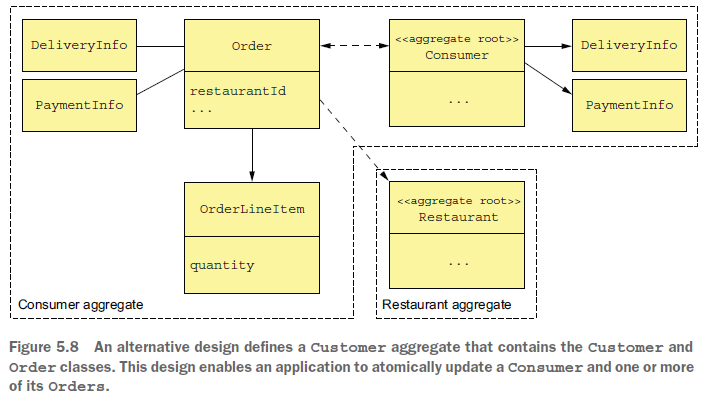
**Or is there?** It turns out that aggregate boundaries are not set in stone. When developing a domain model, you get to choose where the boundaries lie. But like a 20th century colonial power, drawing national boundaries, you need to be careful.

## Aggregate Granularity

When developing a domain model, a key decision you must make is how large to make each aggregate.

* On one hand, **aggregates should ideally be small**. Because updates to each aggregate are serialized, more fine-grained aggregates will increase the number of simultaneous requests that the application can handle, improving scalability. It will also improve the user experience because it reduces the chance of two users attempting conflicting updates of the same aggregate.
* On the other hand, because an aggregate is the scope of transaction, you may need to define **a larger aggregate in order to make a particular update atomic.**

For example, earlier I mentioned how in the FTGO application’s domain model Order and Consumer are separate aggregates. An alternative design is to make Order part of the Consumer aggregate. Figure 5.8 shows this alternative design:



* A benefit of this larger Consumer aggregate is that the application can atomically update a Consumer and one or more of its Orders.
* A drawback of this approach is that it reduces scalability. Transactions that update different orders for the same customer would be serialized. Similarly, two users would conflict if they attempted to edit different orders for the same customer.
* Another drawback of this approach in a microservice architecture is that it is an obstacle to decomposition. The business logic for Orders and Consumers, for example, must be collocated in the same service, which makes the service larger**.**

**Because of these issues, making aggregates as fine-grained as possible is best.**

## Designing Business Logic with Aggregates

In a typical (micro)service**, the bulk of the business logic consists of aggregates**. **The rest of the business logic resides in the domain services** and **the sagas**.

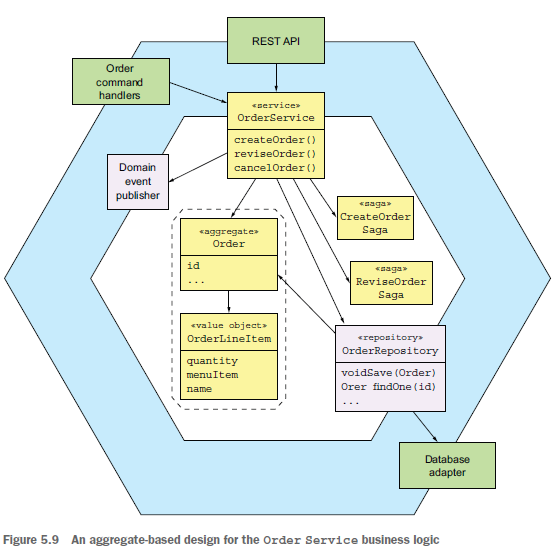
* The sagas orchestrate sequences of local transactions in **order to enforce data consistency**.
* **The services are the entry points into the business logic** and are invoked by inbound adapters.
* A service uses a repository to retrieve aggregates from the database or save aggregates to the database.
* Each repository is implemented by an outbound adapter that accesses the database. Figure 5.9 shows the aggregate-based design of the business logic for the Order Service.

The business logic consists of

1. the Order aggregate
2. the OrderService service class
3. the OrderRepository
4. and one or more sagas.

The OrderService invokes the Order-Repository to save and load Orders.

For simple requests that are local to the service, the service updates an Order aggregate. If an update request spans multiple services, the OrderService will also create a saga, as described in chapter 4.



We’ll take a look at the code—but first, let’s examine a concept that’s **closely related to aggregates: domain events.**

# Publishing Domain Events

In the context of DDD, **a domain event is something that has happened to an aggregate**. **It’s** **represented by a class in the domain model**. **An event usually represents a state change**. Consider, for example, an Order aggregate in the FTGO application. Its state-changing events include **Order Created, Order Cancelled, Order Shipped**, and so forth. An Order aggregate might, **if there are interested consumers, publish one of the events each time it undergoes a state transition.**

## Why Publish Change Events

Domain events are useful because other parties—users, other applications, or other components within the same application—are often interested in knowing about an aggregate’s state changes. Here are some example scenarios:

 **Maintaining data consistency across services using choreography-based sagas**, described in chapter4.

 **Notifying a service that maintains a replica that the source data has changed**. This approach is known as Command Query Responsibility Segregation (CQRS), and it’s described in chapter 7.

 **Notifying a different application via a registered webhook or via a message broker**

**in order to trigger the next step in a business process.**

 **Notifying a different component of the same application in order, for example, to send a WebSocket message to a user’s browser** or update a text database such as ElasticSearch.

 **Sending notifications**—text messages or emails—to users informing them that their order has shipped, their Rx prescription is ready for pick up, or their flight is delayed.

 **Monitoring domain events to verify that the application is behaving correctly.**

 **Analyzing events to model user behavior**.

**The trigger** for the notification in all these scenarios **is the state change of an aggregate in an application’s database.**

## What is a Domain Event?

A *domain event* is **a class** with **a name** formed using **a past-participle verb**. It has **properties that meaningfully convey the event.** Each property **is either a primitive value** or **a value object**. For example, an OrderCreated event class has an orderId property.

A domain event typically also has metadata, such as the event ID, and a timestamp. It might also have the identity of the user who made the change, because that’s useful for auditing. The metadata can be part of the event object, perhaps defined in a superclass. Alternatively, the event metadata can be in an envelope object that wraps the event object. The ID of the aggregate that emitted the event might also be part of the envelope rather than an explicit event property.

The OrderCreated event is an example of a domain event. It doesn’t have any fields, because the Order’s ID is part of the event envelope. The following listing shows the OrderCreated event class and the DomainEventEnvelope class.

interface DomainEvent {}

interface OrderDomainEvent extends DomainEvent {}

class OrderCreated implements OrderDomainEvent {}

class DomainEventEnvelope<T extends DomainEvent> {

private String aggregateType;

private Object aggregateId;

private T event;

...

}

**The DomainEvent interface is a marker interface** that identifies a class as a domain event. OrderDomainEvent is a marker interface for events, such as OrderCreated, which are published by the Order aggregate. **The DomainEventEnvelope is a class that contains event metadata and the event object. It’s a generic class that’s parameterized by the domain event type.**

## Event Enrichment