Unmesh Joshi

318 Followers · About Follow

Aggregate Oriented Microservices

Unmesh Joshi Jun 8, 2018 · 6 min read

In last few years Domain Driven Design has resurrected in the context of MicroServices. A lot of books advocate use of Bounded Contexts decide Microservice boundaries. While Bounded Contexts is good for identifying broader grouping of services, when developing Service APIs, it's generally guided by another very important concept called 'Aggregates'. Aggregates in DDD relate to several other analysis and design techniques in the past, like use cases, naked objects and the well known pattern MVC. But Aggregates make the concept of transactional boundaries explicit. In this write up, we will look at what Aggregates are, how they relate to Use Cases, Naked Objects and MVC. How Services APIs can be guided by Aggregates and then see an example to use Akka Actors to model aggregates.

Aggregates in DDD

Aggregates is one of the important patterns defined in Domain Driven Design. In simple terms, it can be defined as a group of objects which are acted upon by end user transactionally. There are two important points here.

1. The group of objects are always acted upon together and not separately.

2. The actions are transactional. Either updates to all the objects happen or none at all. Other way to say this is to say that all the updates happening to these set of objects always keep the object state consistent. For Example, if we have a Customer object containing a set of Addresses, and for our use cases, any update that happens always

through the Customer object. The decision to make Customer as an Aggregate vs individual Address as an Aggregate is driven by the domain and use cases or user interactions.

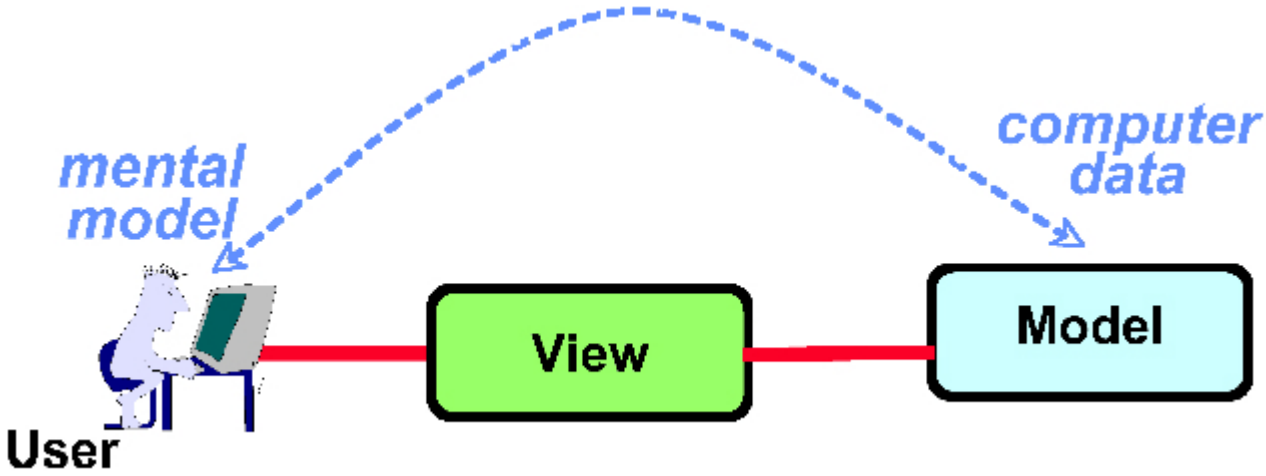
Finding Aggregates

Historically, there were various approaches discussed to help model collection of data elements as Objects which users interact with. This is particularly crucial for knowing the 'boundary' of user interactions. Use Cases helped analysing the software system by looking at end users interacting with the system. That way they help drawing boundary around the set of objects and defining interfaces supporting those actions.

Interestingly, the main focus of well known Model-View-Controller pattern was similar.

It was to help model objects following the end user's mental model. Giving users a perception that they are manipulating the objects directly. This in turn also helps structuring the data elements user interacts with as meaningful Objects which have meaning in user's mental model. This is explained in great details in an article by Trygve Reenskaug and James Coplien.

Direct Manipulation Metaphor From James O. Coplien And Trygve Reenskaug (http://www.artima.com/articles/dci_vision.html)



Following the same principle, but taking it one level further, the 'Naked Objects' approach (https://en.wikipedia.org/wiki/Naked_objects) say that all the business logic

will correctly reflect it.

Aggregates in DDD are conceptually in line with this thinking. They are objects which map to end user's mental model (or Domain Model), with which users interact.

Aggregates put one more important constraint on user actions on objects. The actions also need to be transactional.(Following the ACID properties).

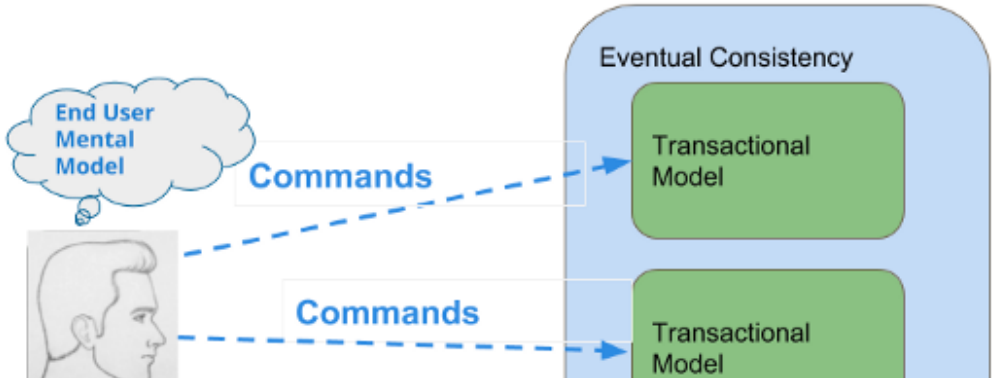
So to find aggregates, we can start with use cases and find objects or object groups user actions map to. The other aspect to make sure is that these actions need to be transactional.

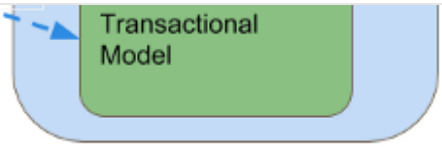
In case a single step in a use case spans multiple aggregates we have to make sure we split those steps to make transactional steps explicit.

Microservices and Aggregates

How do aggregates relate to microservices? When we expose APIs, they will generally be for user actions which are transactional. A bounded context or a subdomain may have multiple aggregates. So a 'service api', can be thought of as a set of operations, each being a command to an aggregate.

This is one of the good guidelines to follow when the question arises, what should be our Service or API boundary?. An API should map to an action on an aggregate which maps to end users mental model. One of the first steps is to clearly know the use cases of the API we are building.



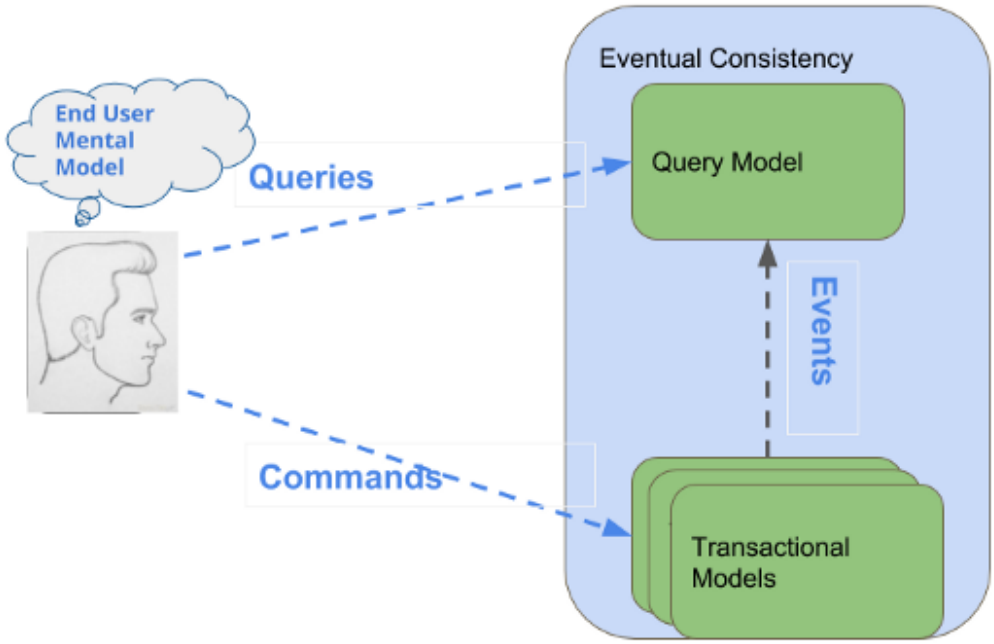


Aggregates, CQRS And Event Sourcing

API based systems use the API as the contract for end users / consumers. In Event Based systems. the “event” is the contract for the consumers. These events are usually “business” events, and need the same kind of design thinking that is applied to APIs. Once we design service APIs as actions on Aggregates, It is also easy to think of events those actions map to. E.g. If ReserveSeats is an action on an Aggregate, the corresponding event can be SeatsReserved. In a system designed this way around Aggregates, it’s easier to implement event collaboration or event sourcing.

Aggregates can guide the design so that we can distinguish user actions(writes) and user queries(reads). Thinking of the API design as user actions and user queries, naturally helps designing the services for CQRS (Command Query Responsibility Segregation).

The arrangement looks something like following



state as series of events instead of doing in place updates of aggregate states. These events can be re-played anytime to recreate state. This can add some complexity to the system, but has many advantages for recovery, playback, rollback, auditing, system evolution, etc.

A Movie Ticket Booking Example

Let's look at a familiar example. Consider a software system for booking movie tickets. A typical use case look like following

Use Case 'Book movie tickets'

1. System shows available Movies
2. User selects specific Movie
3. User selects a 'Show' for a specific movie
4. User 'reserves seats' for the selected Show
5. User enters contact details and confirms Order.
6. System shows payment options to user
7. User makes payment
8. After successful payment, system confirms Order and notifies user.

One of the first candidates for aggregates we can see here is 'Show'. A specific show has a set of seats which user selects. The seat reservation action from user should follow ACID properties. No two users should be able to reserve same seats. User does 'reserve seats' action on this aggregate. So one of our service APIs is going to be /reserve-seats. From service perspective it looks like following

If we want to add event sourcing and CQRS, the arrangement will look like following

Aggregates and Akka Actors

This view of Aggregates fits naturally with concept of Actors provided by frameworks like Akka. Akka provides a framework called Persistent Actors which can be used to implement Aggregates modelling user actions as actor messages handled by transactional objects.

We can implement it with Actors as following

Next Steps

Similar to Show, other user actions can guide to create aggregates like Order, Movie etc.. It's also important to know how to manage workflows which can involve co-ordinating actions on multiple aggregates. There are range of techniques from simple co-ordination through events to full process manager implementation.

Summary

Aggregates provide good guidance to design microservice APIs. They help focusing on transactional boundaries within the system. This helps finding 'seams' to break the system into parts. It is possible to identify aggregates using well known analysis techniques like use cases. A 'service api', can be thought of as a set of operations, each being a command to an aggregate. Frameworks like Akka Persistent Actors provide a good way to model aggregates in code.

Software Architecture Ddd Microservices Akka