(/)

Persisting DDD Aggregates

Last modified: December 4, 2018

by Mike Wojtyna (https://www.baeldung.com/author/michal-wojtyna/)

Persistence (https://www.baeldung.com/category/persistence/)

Spring Data (https://www.baeldung.com/category/persistence/spring-persistence/spring-data/)

1. Overview

In this tutorial, we'll explore the possibilities of persisting DDD Aggregates (https://martinfowler.com/bliki/DDD_Aggregate.html) using different technologies.

2. Introduction to Aggregates

An aggregate is a group of business objects which always need to be consistent. Therefore, we save and update aggregates as a whole inside a transaction.

Aggregate is an important tactical pattern in DDD, which helps to maintain the consistency of our business objects. However, the idea of aggregate is also useful outside of the DDD context.

There are numerous business cases where this pattern can come in handy. **As a rule of thumb, we should consider using aggregates when there are multiple objects changed as part of the same transaction**.

Let's take a look at how we might apply this when modeling an order purchase.

2.1. Purchase Order Example

So, let's assume we want to model a purchase order:

```
class Order {
    private Collection<OrderLine> orderLines;
    private Money totalCost;
    // ...
}
```

```
class OrderLine {
    private Product product;
    private int quantity;
    // ...
}
```

```
class Product {
    private Money price;
    // ...
}
```

These classes form a simple aggregate. Both *orderLines* and *totalCost* fields of the *Order* must be always consistent, that is *totalCost* should always have the value equal to the sum of all *orderLines*.

Now, we all might be tempted to turn all of these into fully-fledged Java Beans. But, note that introducing simple getters and setters in *Order* could easily break the encapsulation of our model and violate business constraints.

Let's see what could go wrong.

2.2. Naive Aggregate Design

Let's imagine what could happen if we decided to naively add getters and setters to all properties on the *Order* class, including *setOrderTotal*.

There's nothing that prohibits us from executing the following code:

```
Order order = new Order();
order.setOrderLines(Arrays.asList(orderLine0, orderLine1));
order.setTotalCost(Money.zero(CurrencyUnit.USD)); // this doesn't look good...
```

In this code, we manually set the *totalCost* property to zero, violating an important business rule. Definitely, the total cost should not be zero dollars!

We need a way to protect our business rules. Let's look at how Aggregate Roots can help.

2.3. Aggregate Root

An *aggregate root* is a class which works as an entry point to our aggregate. **All business operations should go through the root.** This way, the aggregate root can take care of keeping the aggregate in a consistent state.

The root is what takes cares of all our business invariants.

And in our example, the *Order* class is the right candidate for the aggregate root. We just need to make some modifications to ensure the aggregate is always consistent:

```
class Order {
    private final List<OrderLine> orderLines;
    private Money totalCost;

    Order(List<OrderLine> orderLines) {
        checkNotNull(orderLines);
        if (orderLines.isEmpty()) {
            throw new IllegalArgumentException("Order must have at least one order line item");
        }
        this.orderLines = new ArrayList<>(orderLines);
        totalCost = calculateTotalCost();
    }

    void addLineItem(OrderLine orderLine) {
        checkNotNull(orderLine);
        orderLines.add(orderLine);
        totalCost = totalCost.plus(orderLine.cost());
    }

    void removeLineItem(int line) {
        OrderLine removedLine = orderLines.remove(line);
        totalCost = totalCost.minus(removedLine.cost());
    }

    Money totalCost() {
        return totalCost;
    }

    // ...
}
```

Using an aggregate root now allows us to more easily turn *Product* and *OrderLine* into immutable objects, where all the properties are final.

As we can see, this is a pretty simple aggregate.

And, we could've simply calculated the total cost each time without using a field.

However, right now we are just talking about aggregate persistence, not aggregate design. Stay tuned, as this specific domain will come in handy in a moment.

How well does this play with persistence technologies? Let's take a look. **Ultimately, this will help us to choose the right persistence tool for our next project**.

3. JPA and Hibernate

In this section, let's try and persist our *Order* aggregate using JPA and Hibernate. We'll use Spring Boot and JPA (https://search.maven.org/search?q=g:org.springframework.boot%20AND%20a:spring-boot-starter-data-jpa) starter:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

For most of us, this seems to be the most natural choice. After all, we've spent years working with relational systems, and we all know popular ORM frameworks.

Probably the biggest problem when working with ORM frameworks is the simplification of our model design. It's also sometimes referred to as Object-relational impedance mismatch (https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch). Let's think about what would happen if we wanted to persist our *Order* aggregate:

```
@DisplayName("given order with two line items, when persist, then order is saved")
@Test
public void test() throws Exception {
    // given
    JpaOrder order = prepareTestOrderWithTwoLineItems();

    // when
    JpaOrder savedOrder = repository.save(order);

    // then
    JpaOrder foundOrder = repository.findById(savedOrder.getId())
      .get();
    assertThat(foundOrder.getOrderLines()).hasSize(2);
}
```

At this point, this test would throw an exception: *java.lang.IllegalArgumentException: Unknown entity: com.baeldung.ddd.order.Order*. **Obviously, we're missing some of the JPA requirements:**

1. Add mapping annotations
2. *OrderLine* and *Product* classes must be entities or *@Embeddable* classes, not simple value objects
3. Add an empty constructor for each entity or *@Embeddable* class
4. Replace *Money* properties with simple types

Hmm, we need to modify the design of *Order* aggregate to be able to use JPA. While adding annotations is not a big deal, the other requirements can introduce a lot of problems.

3.1. Changes to the Value Objects

The first issue of trying to fit an aggregate into JPA is that we need to break the design of our value objects: Their properties can no longer be final, and we need to break encapsulation.

We need to add artificial ids to the *OrderLine* and *Product*, even if these classes were never designed to have identifiers. We wanted them to be simple value objects.

It's possible to use *@Embedded* and *@ElementCollection* annotations instead, but this approach can complicate things a lot when using a complex object graph (for example *@Embeddable* object having another *@Embedded* property etc.).

Using *@Embedded* annotation simply adds flat properties to the parent table. Except that, basic properties (e.g. of *String* type) still require a setter method, which violates the desired value object design.

Empty constructor requirement forces the value object properties to not be final anymore, breaking an important aspect of our original design. Truth be told, Hibernate can use the private no-args constructor, which mitigates the problem a bit, but it's still far from being perfect.

Even when using a private default constructor, we either cannot mark our properties as final or we need to initialize them with default (often null) values inside the default constructor.

However, if we want to be fully JPA-compliant, we must use at least protected visibility for the default constructor, which means other classes in the same package can create value objects without specifying values of their properties.

3.2. Complex Types

Unfortunately, we cannot expect JPA to automatically map third-party complex types into tables. Just see how many changes we had to introduce in the previous section!

For example, when working with our *Order* aggregate, we'll encounter difficulties persisting *Joda Money* fields.

In such a case, we might end up with writing custom type *@Converter* available from JPA 2.1. That might require some additional work, though.

Alternatively, we can also split the *Money* property into two basic properties. For example *String* for currency unit and *BigDecimal* for the actual value.

While we can hide the implementation details and still use *Money* class through the public methods API, the practice shows most developers cannot justify the extra work and would simply degenerate the model to conform to the JPA specification instead.

3.3. Conclusion

While JPA is one of the most adopted specifications in the world, it might not be the best option for persisting our *Order* aggregate.

If we want our model to reflect the true business rules, we should design it to not be a simple 1:1 representation of the underlying tables.

Basically, we have three options here:

1. Create a set of simple data classes and use them to persist and recreate the rich business model. Unfortunately, this might require a lot of extra work.
2. Accept the limitations of JPA and choose the right compromise.
3. Consider another technology.

The first option has the biggest potential. In practice, most projects are developed using the second option.

Now, let's consider another technology to persist aggregates.

4. Document Store

A document store is an alternative way of storing data. Instead of using relations and tables, we save whole objects. **This makes a document store a potentially perfect candidate for persisting aggregates**.

For the needs of this tutorial, we'll focus on JSON-like documents.

Let's take a closer look at how our order persistence problem looks in a document store like MongoDB.

4.1. Persisting Aggregate Using MongoDB

Now, there are quite a few databases which can store JSON data, one of the popular being MongoDB. MongoDB actually stores BSON, or JSON in binary form.

Thanks to MongoDB, we can store the *Order* example aggregate *as-is*.

Before we move on, let's add the Spring Boot MongoDB (https://search.maven.org/search?q=g:org.springframework.boot%20AND%20a:spring-boot-starter-data-mongodb) starter:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

Now we can run a similar test case like in the JPA example, but this time using MongoDB:

```
@DisplayName("given order with two line items, when persist using mongo repository, then order is save
@Test
void test() throws Exception {
    // given
    Order order = prepareTestOrderWithTwoLineItems();

    // when
    repo.save(order);

    // then
    List<Order> foundOrders = repo.findAll();
    assertThat(foundOrders).hasSize(1);
    List<OrderLine> foundOrderLines = foundOrders.iterator()
      .next()
      .getOrderLines();
    assertThat(foundOrderLines).hasSize(2);
    assertThat(foundOrderLines).containsOnlyElementsOf(order.getOrderLines());
}
```

What's important – we didn't change the original *Order* aggregate classes at all; no need to create default constructors, setters or custom converter for *Money* class.

And here is what our *Order* aggregate appears in the store:

```
{
  "_id": ObjectId("5bd8535c81c04529f54acd14"),
  "orderLines": [
    {
      "product": {
        "price": {
          "money": {
            "currency": {
              "code": "USD",
              "numericCode": 840,
              "decimalPlaces": 2
            },
            "amount": "10.00"
          }
        }
      },
      "quantity": 2
    },
    {
      "product": {
        "price": {
          "money": {
            "currency": {
              "code": "USD",
              "numericCode": 840,
              "decimalPlaces": 2
            },
            "amount": "5.00"
          }
        }
      },
      "quantity": 10
```

```
      }
    ],
    "totalCost": {
      "money": {
        "currency": {
          "code": "USD",
          "numericCode": 840,
          "decimalPlaces": 2
        },
        "amount": "70.00"
      }
    },
    "_class": "com.baeldung.ddd.order.mongo.Order"
}
```

This simple BSON document contains the whole *Order* aggregate in one piece, matching nicely with our original notion that all this should be jointly consistent.

Note that complex objects in the BSON document are simply serialized as a set of regular JSON properties. Thanks to this, even third-party classes (like *Joda Money*) can be easily serialized without a need to simplify the model.

4.2. Conclusion

Persisting aggregates using MongoDB is simpler than using JPA.

This absolutely doesn't mean MongoDB is superior to traditional databases. There are plenty of legitimate cases in which we should not even try to model our classes as aggregates and use a SQL database instead.

Still, when we've identified a group of objects which should be always consistent according to the complex requirements, then using a document store can be a very appealing option.

5. Conclusion

In DDD, aggregates usually contain the most complex objects in the system. Working with them needs a very different approach than in most CRUD applications.

Using popular ORM solutions might lead to a simplistic or over-exposed domain model, which is often unable to express or enforce intricate business rules.

Document stores can make it easier to persist aggregates without sacrificing model complexity.

The full source code of all the examples is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/ddd).

▲ newest ▲ oldest ▲ most voted

Guest

Waldemar (https://biller.digital)

Hi Mike, It's great that you wrote about this topic! Just some things to consider: 1. If totalCost and OrderLines have to be consistent, why not compute the totalCost by iterating over the OrderLines and sum up all prices in a sophisticated getter? This is the absolute consistency! Within this getter you could also do the transformation to the Money class. 2. The Aggregate Root is not the only thing that has to be an entity. Aggregate Root means nothing but the root of the object graph which you retrieve from the database either by one of the find*/get*-Methods or… Read more »

+ 1 −

1 year ago

(https://www.baeldung.com/author/michal-wojtyna/)

Author

Mike Wojtyna (http://buildmysoftware.pro)

Hi Waldemar! Thanks for your detailed comment. Here are my answers: 1. That's exactly what I mentioned in the article here: "As we can see, this is a pretty simple aggregate. And, we could've simply calculated the total cost each time without using a field. However, right now we are just talking about aggregate persistence, not aggregate design" and here "While we can hide the implementation details and still use Money class through the public methods API, the practice shows most developers cannot justify the extra work and would simply degenerate the model to conform to the JPA specification instead."... Read more »

+ 0 −

1 year ago

Comments are closed on this article!

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
JOBS (/TAG/ACTIVE-JOB/)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)